Projecte de Final de Carrera

Enginyeria Informàtica

Estudi comparatiu de l'algorisme basat en una colònia artificial d'abelles

Autor:
Joan T. Matamalas Llodrà

Director:
Dr. David Isern Alarcón

13 de juny de 2012

Als meus pares pel seu suport.

Al Dr. Isern per la seva paciència i consells.

Índex

Agraïments **1**

Índex de figures **4**

Índex de taules **5**

1 Introducció **6**
1.1 Resum 6
1.2 Motivació 7
1.3 Objectius 7
1.4 Tasques 7
1.4.1 Relació de tasques 8
1.4.2 Planificació temporal 10

2 Fonaments de la computació evolutiva **11**
2.1 Algoritmes 11
2.1.1 Algoritmes evolutius (EA) 12
2.1.2 Intel·ligència d'eixam (SI) 16
2.2 Història 17
2.3 Camps d'utilització 19

3 Descripció i fonaments dels algoritmes utilitzats **21**
3.1 Algoritmes genètics (GA) 21
3.1.1 Fonaments 21
3.1.2 Tipus de representació utilitzat 22
3.1.3 Metodologia de selecció dels 'pares' 22
3.1.4 *Crossover* i generació del *offspring* 24
3.1.5 Procés de mutació 25
3.1.6 Selecció dels individus que formaran la nova generació 26
3.2 Evolució diferencial (DE) 27
3.2.1 Fonaments 27
3.2.2 Tipus de representació utilitzat 27
3.2.3 Procés de mutació 27
3.2.4 Mètode de *crossover* 28
3.2.5 Selecció de la propera generació 28
3.3 Optimització amb eixam de partícules (PSO) 28
3.3.1 Fonaments 28

3.3.2 Topologia 29
3.3.3 Representació dels paràmetres 30
3.3.4 Càlcul del desplaçament 30
3.3.5 Velocitat màxima 31
3.3.6 Constricció simple (χ) 31
3.4 Colònia artificial d'abelles (ABC) 31
3.4.1 Fonaments 31
3.4.2 Funcionament de l'algoritme 33

4 Descripció de l'experiment **35**
4.1 Metodologia 35
4.1.1 Mesures 35
4.1.2 Condicions de fi d'execució 36
4.1.3 Funcions a optimitzar 36
4.1.4 Metodologia utilitzada 40

5 Resultats **42**
5.1 Resultats de les execucions 42
5.1.1 Anàlisi dels estadístics 49
5.2 Comparativa dels resultats 51

6 Conclusions **54**
6.1 Possibles millores 55

A Detalls de l'implementació del prototipus **56**
A.1 Programari utilitzat 56
A.2 Maquinaria utilitzada 57
A.3 Estructura del prototipus 57
A.4 Implementació de l'experiment 58
A.5 Implementació de les funcions d'avaluació 62
A.6 Implementació dels EC 63
A.6.1 Implementació del GA 65
A.6.2 Implementació del DE 66
A.6.3 Implementació del PSO 67
A.6.4 Implementació del ABC 70

Glossari **73**

Llistat d'acrònims **74**

Bibliografia **75**

Índex de figures

2.1 Cromosomes resultants de la meiosi 13
2.2 Workflow dels EA 15
2.3 Sistema funcional 19
2.4 Braç del satèl·lit 20

3.1 Mètodes de mostreig 24
3.2 Exemple de l'encreuament per un punt 25
3.3 Topologies 29
3.4 Comportament de les abelles de la mel 33

4.1 Representació 3D de les funcions d'avaluació 38

5.1 Mitjana d'errors absoluts 53

A.1 UML de classes que interactuen a l'experiment 58
A.2 Opcions disponibles per a l'execució de l'experiment 59
A.3 Barra de progrés de l'execució 61
A.4 Estadístics de la funció Sphere 61
A.5 Resultat d'una comparativa 62

Índex de taules

1.1 Planificació temporal 10

2.1 Aproximació a la resolució de problemes mitjançant conceptes evolutius . 12

4.1 Funcions d'avaluació utilitzades 37
4.2 Constants utilitzades per a la funció Shekel 40

5.1 Resultats per a la funció sphere 42
5.2 Resultats per a la funció step 43
5.3 Resultats per a la funció sumsquares 43
5.4 Resultats per a la funció colville 43
5.5 Resultats per a la funció Zakharov 44
5.6 Resultats per a la funció Schwefel 2p22 44
5.7 Resultats per a la funció Himmelblau 45
5.8 Resultats per a la funció Rastrigin 45
5.9 Resultats per a la funció Schwefel 46
5.10 Resultats per a la funció Michalewicz 5 46
5.11 Resultats per a la funció schaffer 47
5.12 Resultats per a la funció Shubert 47
5.13 Resultats per a la funció Griewank 48
5.14 Resultats per a la funció Ackley 48
5.15 Resultats per a la funció Shekel10 48
5.16 Resum dels resultats 49
5.17 Comparativa ABC vs GA 51
5.18 Comparativa ABC vs DE 52
5.19 Comparativa ABC vs PSO 52
5.20 Resum de comparatives 53

Capítol 1

Introducció

1.1 Resum

La finalitat d'aquest projecte és la realització d'un estudi comparatiu de l'algoritme basat en una colònia artificial d'abelles, Artificial Bee Colony (ABC), comparat amb un conjunt d'algoritmes fonamentats en el paradigma de la computació evolutiva. S'utilitzarà l'eficàcia[1] a l'hora d'optimitzar diverses funcions com a mesura comparativa.

Els algorimtes amb els quals es comparara l'algoritme ABC són:

- Algoritmes genètics[2].
- Evolució diferencial[3].
- Optimització amb eixam de partícules[4].

Els primers dos algoritmes, GA i DE, basen el seu funcionament en els principis de l'evolució genètica, utilitzant els mecanismes de reproducció, mutació i selecció per guiar el procés de cerca. En canvi l'algoritme ABC i el PSO es basen en el comportament social per optimitzar els paràmetres de la funció.

En concret, l'ABC, l'algoritme a comparar, es basa en el comportament social de les abelles a l'hora de cercar zones riques en nèctar i en la manera en què aquestes interactuen per compartir l'informació.

[1] La metodologia utilitzada per avaluar l'eficàcia s'especificarà quan es descriguin els detalls de l'experiment, apartat 4.1.

[2] Genetic Algorithm (GA)

[3] Differential Evolution (DE)

[4] Particle Swarm Optimization (PSO)

1.2 Motivació

A l'actualitat existeixen nombrosos problemes que, donada la seva complexitat, són difícilment resolubles en un temps raonable mitjançant computadors. El problema del viatjant de comerç o el de colorejar un graf en són només uns exemples.

Descrita aquesta problemàtica, pot ser més interessant trobar una solució prou bona en un temps acceptable que no pas intentar resoldre de manera determinista el problema. Els algoritmes basats en computació evolutiva estan especialment indicats per aquesta tasca.

Per tant, la principal motivació per a la realització d'aquest Projecte Final de Carrera (PFC) és l'estudi del funcionament d'algoritmes basats en sistemes biològics per a la resolució de problemes de cerca de complexitat elevada. S'agafarà com a referència l'estudi comparatiu publicat per Dervis Karaboga [32], pare de l'algoritme ABC, on es compara l'eficàcia d'aquest amb altres algoritmes basats en el paradigma de la computació evolutiva.

A part de la realització de la comparativa, s'intentaran exposar els principals elements que sustenten el funcionament dels algoritmes així com donar-ne un visió històrica de la seva evolució.

1.3 Objectius

Els objectius que es pretenen aconseguir són:

- Exposar els fonaments de la computació evolutiva.
- Comparar l'eficàcia dels ABC envers els algoritmes anterirorment esmentats.
- Detallar les bases funcionals dels diferents algoritmes utilitzats.
- Raonar sobre la selecció del valor dels paràmetres configurables dels diferents algoritmes i els efectes que la seva elecció té sobre el seu funcionament.
- Implementar el prototipus que executarà l'experiment.
- Exposar les conclusions extretes de la realització de l'experiment.

1.4 Tasques

L'elaboració d'aquest PFC té una part teòrica i una part pràctica. Per tant, la metodologia utilitzada per la seva realització difereix de la tradicionalment utilitzada en els cicles de producció de programari.

S'ha dividit aquest projecte en diferents parts, com veurem tot seguit (apartat 1.4.1), de manera que la documentació, disseny i implementació de cada una d'aquestes es pot fer de manera independent.

1.4.1 Relació de tasques

Aquest PFC s'ha desglossat en les següent tasques:

1. Definir els objectius i motivació del PFC.
2. Desglossar i planificar les tasques a realitzar.
3. Fonaments de la computació evolutiva.
   (a) Documentació.
   (b) Redacció del contingut.
   (c) Revisió del contingut
4. Estat de l'art
   (a) Documentació.
   (b) Redacció del contingut.
   (c) Revisió del contingut
5. Definició de l'experiment.
   (a) Documentar metodologia.
   (b) Redactar els procediments.
   (c) Definir el mecanisme comparatiu.
   (d) Documentar els diferents tipus de funcions.
   (e) Seleccionar les funcions a utilitzar.
   (f) Definir els paràmetres d'aquestes funcions.
   (g) Revisió del contingut.
   (h) Implementar les funcions a optimitzar.
   (i) Disseny del prototipus
   (j) Implementar el nucli del prototipus.
   (k) Fase de proves
6. Algoritmes genètics, GA.
   (a) Documentar els fonaments.
   (b) Disseny de l'algoritme.
      i. Definir el tipus de representació utilitzat (binari, nombre reals...)
      ii. Seleccionar el sistema de selecció 'paterna'.
      iii. Definir el mètode encreuament utilitzat per combinar el 'codi genètic' dels 'pares' i produir l'*offspring*.
      iv. Establir el mètode de mutació utilitzat.
      v. Fixar el mètode de selecció dels membres de la nova generació.

vi. Seleccionar els paràmetres més adients per l'experiment a realitzar (rati de mutació, rati de la 'mating pool'...).

(c) Implementar l'algoritme.

(d) Fase de proves.

(e) Redactar els continguts.

(f) Revisió dels continguts.

7. Evolució diferencial, DE.

   (a) Documentar els fonaments.

   (b) Disseny de l'algoritme.

      i. Definir el tipus de representació utilitzat (binari, nombre reals...)

      ii. Establir el mètode de mutació utilitzat.

      iii. Definir el mètode encreuament utilitzat per combinar el 'codi genètic' dels 'pares' i produir l'*offspring*.

      iv. Fixar el mètode de selecció dels membres de la nova generació.

      v. Seleccionar els paràmetres més adients per l'experiment a realitzar.

   (c) Implementar l'algoritme.

   (d) Fase de proves.

   (e) Redactar els continguts.

   (f) Revisió dels continguts.

8. Optimització amb eixam de partícules, PSO.

   (a) Documentar els fonaments.

   (b) Disseny de l'algoritme.

      i. Definir el tipus de representació utilitzat (binari, nombre reals...)

      ii. Fixar la topologia més adient per resoldre l'experiment

      iii. Seleccionar els paràmetres més adients per l'experiment a realitzar (parametres de control, el pes de l'inèrcia...).

   (c) Implementar l'algoritme.

   (d) Fase de proves.

   (e) Redactar els continguts.

   (f) Revisió dels continguts.

9. Colònia artificial d'abelles, ABC.

   (a) Documentar els fonaments.

   (b) Disseny de l'algoritme.

      i. Definir el tipus de representació utilitzat (binari, nombre reals...)

ii. Seleccionar els paràmetres més adients per l'experiment a realitzar (paràmetres de control, el pes de l'inèrcia...).

(c) Implementar l'algoritme.

(d) Fase de proves.

(e) Redactar els continguts.

(f) Revisió dels continguts.

10. Resultats.

    (a) Execució del prototipus.

    (b) Redacció dels resultats obtinguts.

    (c) Interpretació dels resultats.

    (d) Redacció de les conclusions.

    (e) Revisió dels continguts elaborats.

11. Presentació.

    (a) Disseny de la presentació.

    (b) Elaboració de la presentació.

    (c) Revisió de la presentació.

12. Revisió de tot el contingut.

13. Entregar la documentació.

1.4.2 Planificació temporal

La realització d'aquest projecte té tres fites temporals definides:

- **PAC 1:** 10 de març.
- **PAC 2:** 14 d'abril.
- **PAC 3:** 19 de maig.
- **Entrega final:** 13 de juny.

Utilitzant l'enumeració de l'apartat anterior, 1.4.1, la planificació de les tasques seria la que segueix:

Fita	**Tasques**
PAC 1	1 i 2
PAC 2	3, 4, 5, 6 i 7
PAC 3	8, 9 i 10
Entrega final	11, 12 i 13

Taula 1.1: Planificació temporal

Capítol 2

Fonaments de la computació evolutiva

2.1 Algoritmes

Kennedy i Eberhart [19, pag. 142] expliquen que els algoritmes de computació evolutiva difereixen dels utilitzats tradicionalment per a la cerca i optimització principalment en 3 punts:

- Utilitzant una població de punts (solucions candidates) en el procés de cerca.
- Utilitzant directament informació sobre l'aptitud d'una solució candidata en lloc d'utilitzar funcions derivades.
- Utilitzant mètodes estocàstics en lloc de deterministes.

Com s'acaba d'esmentar, els algoritmes utilitzats en computació evolutiva tenen un origen estocàstics, el seu funcionament depèn de l'atzar. Aquest fet és important ja que:

> [...]Probabilistic choices allow creative, innovative exploration of new possibilites, especially in a changing environment [19, pag. 12]

Segons Gregory Bateson [10], els organismes biològics s'adapten realitzant ajustos dins el que ell anomena els 2 grans sistemes estocàstics: l'evolució i la ment. Tenint present aquests dos grans sistemes, a continuació s'exposarà una classificació[1], arbitraria, de les diferents tècniques que formen part d'aquesta disciplina:

- Algoritmes Evolutius (Evolutionary Algorithms (EA)):
  - Algoritmes genètics (GA)
  - Programació genètica
  - Programació evolutiva
  - Estratègia evolutiva
  - Evolució diferencial (DE)
  - ...

[1] Utilitzada per la Viquipèdia [2]

- Intel·ligència d'eixam(Swarm Intelligence (SI)):
  - Optimització utilitzant colònies de formigues
  - PSO
  - ABC
  - ...

Els EA utilitzen patrons típics de la genètica mentre que els SI utilitzen patrons relacionats amb el comportament social. A continuació es descriuran els fonaments generals d'aquestes dues branques.

2.1.1 Algoritmes evolutius (EA)

Fonaments biològics

L'any 1859 es va publicar *L'origen de les especies* [16]. En aquest llibre, Charles Darwin teoritzava sobre els mecanismes que afavoreixen la diversitat biològica, on el concepte de selecció natural juga un paper transcendental. L'idea principal rau en el fet que donada una població inicial d'individus i un medi on habiten, els individus que millor s'adaptin a aquest entorn tendran més possibilitats de sobreviure i reproduir-se, en altres paraules, sobrevivien els més aptes. Això ja dóna una primera aproximació a la manera d'afrontar un problema des de el punt de vista evolutiu, taula 2.1 [20, pag. 2].

Evolució	**Resolució de problemes**
Entorn	Problema
Individu	Solució candidata
Aptitud	Qualitat de la solució candidata

Taula 2.1: Aproximació a la resolució de problemes mitjançant conceptes evolutius

S'ha de remarcar que l'adaptabilitat es basa en determinada mesura en el fenotip[2] de cada individu. Així, determinats trets fenotípics que afavoreixen l'adaptació al medi tenen més possibilitats de ser heretats per les futures generacions. D'altre banda, trets que no contribueixen a l'adaptació al medi, tenen tendència a desaparèixer amb el pas de les generacions.

Des de un punt de vista microscòpic, la evolució es guiada per la genètica. Cada individu té un genotip que codifica el seu fenotip. El fragment bàsic d'informació genètica funcional és el gen. Un gen pot afectar a més d'un tret fenotípic, pleiotropia, o diversos gens poden determinar un mateix tret fenotípic, poligenia. El ha de quedar clar és que sempre una variació en el fenotip està determinada per una variació en el genotip [20].

[2]Característiques físiques i del comportament

El conjunts de gens que formen el genotip està organitzat en diferents cromosomes. Les formes més complexes de vida contenen dues copies semblants de cada cromosoma, una heretada del pare i una de la mare, contingudes dins el nucli de les anomenades cèl·lules diploides. En el cas dels humans, el genotip està estructurant en 23 parells de cromosomes.

No totes les cèl·lules dels organismes contenen dues copies d'un mateix cromosoma. Les cèl·lules haploides, com els gàmetes[3], només tenen una versió de cada cromosoma, la qual es complementarà durant la fecundació amb la versió heretada de l'altre progenitor. Aquest tipus de cèl·lules, com es mostrara tot seguit, juga un paper important a l'hora d'augmentat la diversitat genètica.

Els dos generadors principals de diversitat genètica són les mutacions i la combinació de cromosomes. Les mutacions introdueixen canvis aleatoris en les seqüències d'Àcid desoxiribonucleic (ADN) que composen el genotip, aquestes mutacions poden ser de cinc tipus depenen de l'efecte que tingui sobre el codi: Substitució, delació, inversió, multiplicació i inserció [13, pag. 26].

La combinació de cromosomes té lloc durant la generació dels gàmetes, procés anomenat meiosi. En aquest procés, de cada parell cromosòmic és generen 4 versions diferents de les quals 1 serà transmesa via sexual al zigot: Un cromosoma idèntic al cromosoma matern, un idèntic al patern i dues combinacions complementaries dels dos anteriors, figura 2.1. Aquestes combinacions es generen encreuant en un punt aleatori de la cadena d'ADN els cromosomes patern i matern.

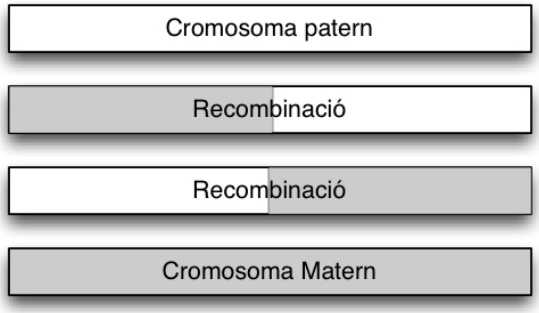


Figura 2.1: Cromosomes resultants de la meiosi

Aquesta encreuament, o *crossover*, a diferència del ques'ha exposat abans per els organismes diploides, en els algoritmes genètics sol tenir lloc en el moment de la generació de la descendència encreuant directament el codi matern amb el patern.

[3]Esperma i òvuls

Funcionament dels EA

Un cop exposats els fonaments biològics en el que es basen els EA, a continuació s'explicaran els elements que els composen. Els més importants són [20, pag. 18]:

- **Representació:** Donat un problema, les solucions candidates d'aquest dins el seu context original (valors enters, permutes d'elements...) formen el que es podria dir el 'fenotip' del problema. Anàlogament al que s'ha explicat anteriorment (secció 2.1.1), les operacions realitzades per els EA funcionen a nivell genètic per tant, es necessita un sistema que permeti codificar els fenotips utilitzant un determinat codi genètic.
  Es a dir, les solucions a un problema poden estar expressades en nombres enters, e.g. 14, i la seva codificació genètica, sobre la que funcionen els EA, pot estar representat en binari '1110'. L'elecció d'un o altre sistema de representació depèn de les operacions genètiques que s'hagin d'utilitzar. El que ha de quedar clar és que la representació del fenotip, no té perquè ser igual a la representació del genotip.

- **Funció d'avaluació:** Aquesta funció serveix per mesurar la qualitat de les solucions candidates, parlant en termes evolutius, determina el grau d'adaptabilitat d'una individu davant l'entorn.

- **Població:** Aquest concepte s'utilitza per descriure el nombre d'individus, solucions candidates, que hi ha en un determinat moment de l'execució de l'algoritme. Hi ha operacions, com la selecció dels pares i la selecció dels supervivents, que s'executen sobre aquest element. Tot i que pot variar depenent del tipus d'algoritme utilitzat, el nombre d'individus que conformen la població es sol mantenir constant durant tota l'execució.

- **Mecanisme de selecció paterna:** Els millors individus de la població, són seleccionats per reproduir-se i generar nous membres.

- **Operadors que augmenten la diversitat:** La seva funció és crear nous individus a partir dels anteriors. Aquest operadors no funcionen tenguen en compte tota la població, es centren en els individus. Els tipus d'operadors es poden dividir depenent de la seva aritat [20, pag. 21]:

  - **Mutació:** Aquest operador s'aplica només a un individu, operador unari, i produeix una versió modificada del mateix. L'utilització d'aquest operador pot variar depenent de l'algoritme utilitzat però la seva finalitat és sempre la mateixa, introduir diversitat 'genètica' dins la població i d'aquesta manera explorar noves zones dins l'espai de solucions.
  - **Encreuament:** Es tracte d'un operador binari, opera sobre dos individus, i el seu objectiu és combinar l'informació de dos individus per produir-ne un de nou. El procés pot variar depenent de la representació utilitzada i l'algoritme allà on s'utilitzi, però essencialment el seu funcionament és intercanviar fragments aleatoris del codi genètic dels pares.

- **Mecanisme de selecció dels supervivents:** El funcionament d'aquest operador és anàleg al funcionament del mecanisme de selecció paterna però en aquest cas, com

es veurà més endavant, s'utilitza en una fase diferent de l'execució de l'algoritme. El seu objectiu és seleccionar els membres que formaran la nova generació de la població entre els de la generació anterior i els recentment produïts tenint en compte la seva adaptabilitat. S'ha de dir que hi ha alguns casos on, a part de l'adaptabilitat, també és té en compte l'*edat*[4] a l'hora de decidir qui passa a la nova generació.

- **Inicialització:** Aquest operador s'utilitza a l'inici de l'execució de l'algoritme i la seva finalitat és generar individus de manera aleatòria per omplir la població.

- **Fi:** Sobre els criteris de fi d'execució, depenen molt de les prioritats que es tinguin: Un resultat precís, a costa del temps d'execució; un resultat ràpid, a costa de la precisió del resultat... Depenent de la prioritats és poden definir una serie de condicions fi d'execució [20]:

  - Màxim temps d'execució.
  - Nombre de cicles d'execució.
  - El valor d'aptitud està per davall d'un determinat llindar durant una serie de cicles d'execució.
  - La diversitat de la població cau per davall d'un predeterminat valor.

Per exemplificar el funcionament bàsic d'aquests tipus d'algoritmes i el moment on intervenen cada un d'aquests elements, Eiben [20, pag. 17] proposa un diagrama, figura 2.2, on es detalla l'esquema general del seu funcionament.

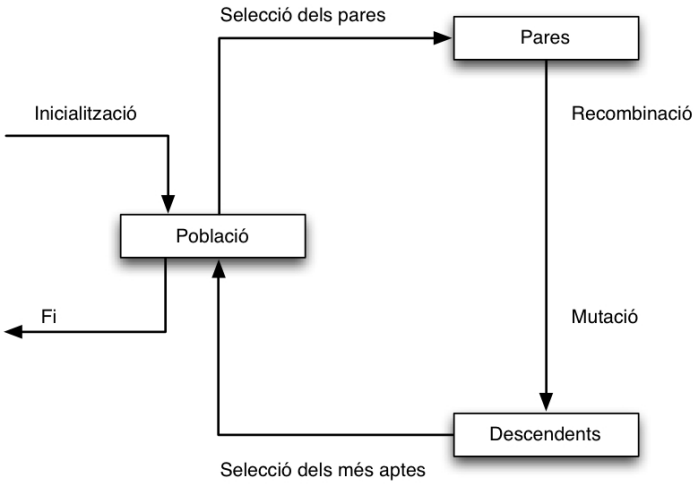


Figura 2.2: Workflow dels EA

[4]Nombre de cicles que un determinat individu ha estat present dins la població

2.1.2 Intel·ligència d'eixam (SI)

El 1989 Beni i Wang introduïren el concepte SI [11] quan estaven estudiant el comportament dels autòmats cel·lulars:

> The cellular automation (CA) is a very simple virtual machine that results in complex, even lifelike, behavior. In the most common onedimensional, binary versions, a 'cell' is a site on a string of ones and zeroes (a *bitstring*): a cell can exist in either of two states, representet as zero or one. The state of a cell at the next time step is determined by its state at the current time and by the state of cells in its neighborhood. A neighborhood usually comprises some number of cells lying on each side of the cell in question. [19, pag. 20]

Fins ara quan s'ha parlat d'adaptació s'ha fet des del punt de vista de genètic o natural, però l'adaptació, com s'acaba de mostrar, també es pot mirar des del punt de vista social, on la *intel·ligència emergeix* de la interacció social dels individus, a priori simples, que formen la població.

Exemples d'aquesta *intel·ligència* emergent, es poden trobar en part de les estructures socials presents a la naturalesa. Com a exemple podem posar les formigues i les abelles, les quals individualment són agents molts simples, amb un patrons de comportament perfectament deterministes que responen els diversos estímuls presents a l'entorn, però que quan interactuen entre si, formen patrons de comportament altament complexos capaços d'adaptar-se a canvis a l'entorn.

Hi ha diversos tècniques que, utilitzant aquesta metàfora, s'han descrit per fer front a problemes de cerca. A continuació es detallaran les més rellevants:

- **Ant colony optimization** [15]: Utilitzant patrons de comportament presents a les colònies de formigues, aquest algoritme guia el procés de cerca desplaçant les *formigues* basant-se en el rastre de *feromones* que els seus congèneres han deixat per l'espai de cerca en iteracions prèvies.

- **Artificial bee colony (ABC)** [31]: Aquest serà un dels algoritmes utilitzats en aquesta pràctica i es bassa en comportament de les abelles de la mel a l'hora de compartir informació sobre la localització de fonts d'aliment.

- **Particle swarm optimization (PSO)** [35]: inicialment concebut per simular el comportament social com una representació estilitzada del moviment d'organismes dins una estol d'ocells o un banc de peixos.

- **Cuckoo search** [48]: És basa en el comportament parasitari que tenen els cucuts a l'hora de deixar els seus ous a nius d'altres ocells perquè siguin aquests qui els incubin.

2.2 Història

L'idea d'utilitzar operacions evolutives a l'hora de resoldre un problema no és nova, de fet les primeres referències a aquest tipus de algoritmes daten dels inicis de l'era de la computació, els anys quaranta. Va ser un dels pares d'aquest camp, Alan Turing, qui en un treball presentat a l'any 1948 diu:

> [...]There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value. [47]

Tot i que aquesta només és una primera idea, no fa cap referència al mètode a seguir, si que és una de les primeres aparicions del concepte de cerca guiada per evolució. A un celebre treball posterior [46], Turing deixa veure els elements necessàris per guiar aquesta cerca:

> [...]We have thus divided our problem into two parts. The child programme and the education process. These two remain very closely connected. We cannot expect to find a good child machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications:
>
> - Structure of the child machine = hereditary material
> - Changes of the child machine = mutation
> - Natural selection = judgment of the experimenter

Els primers treballs experimentals duits a terme daten de final de la dècada dels cinquanta, i la seva finalitat no era l'utilització d'un mètode evolutiu per resoldre problemes de combinatòria, sinó que aquests experiments intentaven fer simulacions sobre sistemes genètics naturals. S'ha de remarcar l'importància que entre aquests va tenir A.S. Fraser [23] ja que va posar les bases que molts investigadors posteriorment seguirien.

No va ser fins l'any 1962, quan Bremermann va conduir un experiment d'optimització utilitzant mètodes evolutius [12]. Aquest primer experiment defineix un primer prototipus del que avui es coneixen com EA. De totes maneres, era una primera aproximació, i com l'autor reconeix en el propi treball:

> Evolution is a fact of life. It has gone a long way to explain biological phenomena. I hope I have demonstrated that there still is a lot to be learned about it[...]

A partir d'aquest moment sorgeixen diferents àrees d'estudi que treballen utilitzant conceptes evolutius per a resoldre determinats tipus de problemes, les quals varen anar desenvolupant-se de manera independent fins a ben entrada la dècada dels noranta. La literatura dóna especial rellevància a aquestes quatre:

- Algoritmes genètics (GA).

- Programació evolutiva.
- Estratègies evolutives.
- Programació genètica.

El concepte d'algoritme genètic no fou introduït fins 1967 quan Bagley [7] els utilitzà per trobar el conjunt de paràmetres adequat per jugar al Hexapawn, un joc que utilitza un tauler de escacs de tres per tres on cada jugador comença utilitzant tres peons.

Tot i que no va ser el primer a utilitzar el concepte, John H. Holland és considerat per molts el pare dels GA. Holland, que va ser estudiant de McCarthy i Weiner i el primer que va obtenir un Ph.D. en computació als EUA, ja a principis dels seixanta parlava de sistemes auto-adaptatius [29]. No va ser però fins l'any 1975, data de publicació del llibre *Adaptation in Natural and Artificial Systems* [30], quan l'interès en el camp augmentà de forma significativa convertint-se en un dels referents en sistemes adaptatius i comença a utilitzar-se en diferents entorns. Per exemple Goldberg, un estudiant de Holland, utilitza aquests algoritmes per minimitzar el consum en les centrals de bombeig de gas [26] [27].

Els algoritmes basats en programació evolutiva foren introduïts, també als EUA, per Fogel, Owens i Walsh a mitjans dels seixanta [22]. L'operació clau en aquest tipus d'algoritmes és la mutació, ja que és l'únic operador estocàstic que utilitza, no hi ha emparellament d'individus. El mateix Fogel explica les diferències entre els GA i la programació evolutiva [19, pag. 139] [21]:

> The procedure abstracts evolution as a top-down process of adaptive behavior, rather than a bottom-up process of adaptive genetics. It is argued that this approach is more appropriate because natural selection does not act on individual components in isolation, but rather on the complete set of expressed behaviors of an organism in light of its interaction with its environment.

Una visió similar sobre el procés d'optimització va ser la que va motivar a Ingo Rechenebrg and Hans-Paul Schwefel a utilitzar l'evolució per optimitzar la resistència al vent d'una superfície [41], i dóna lloc al que es coneixen com a algoritmes basats en estratègies evolutives. Donat que és basen en una metàfora similar, a dia d'avui es fa difícil distingir entre algoritmes basats en programació evolutiva i els basats en estratègies evolutives [3].

La darrera gran àrea, la programació genètica, està més enfocada als problemes d'aprenentatge que no pas als problemes d'optimització [20, pag. 101] i el seu funcionament rau en evolucionar directament el programa per adaptar-lo al problema a solucionar. Fou un dels alumnes de Holland, John Koza, qui és considerat el gran valedor d'aquesta categoria d'algoritmes [37]. Precisament aquesta proximitat amb Holland i els seus mètodes fa que molts considerin la programació genètica una especialització dels GA [19, pag. 135].

Independentment als algoritmes exposats, tots ells basats en l'evolució biològica, Beni i Wang[11] varen encunyar el terme *Intel·ligència en eixam* (SI) quan estaven treballant amb autòmats cel·lulars. Aquest camp es basa en agents molts bàsics, el funcionament

del qual està determinat per regles molt simples, on la seva interacció *social* condueix a l'aparició d'un comportament *intel·ligent*.

No va ser fins a la realització al 1994 del congrés *IEEE World Congress on Computational Intelligence*, el qual allotjava el primer *IEEE Conference on Evolutionary Computation*, quan es varen anar unificant les diferents implementacions baix un mateix camp, la computació evolutiva.

2.3 Camps d'utilització

La computació evolutiva és una branca de la informàtica destinada a la resolució de problemes de combinatòria. Segons exposa Eiben [20, pag. 8-9], aquests problemes es poden representaran utilitzant el diagrama de la figura 2.3 el qual conté els següents components: entrada, model i sortida. Depenent de quin sigui el component que es desconeix podem distingir 3 tipus de problemes.

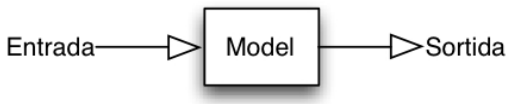


Figura 2.3: Sistema funcional

- **Optimització *(Es desconeix l'entrada)*:** L'objectiu d'aquests tipus de problemes és trobar els valors d'entrada adequats per tal d'obtenir el valor de sortida desitjat. Dels 3 tipus de problemes que es mostraran aquest és el que compte amb una major trajectòria a l'hora d'utilitzar computació evolutiva per a la seva resolució[5]. El problema del viatjant de comerç és d'aquest tipus i donat que la seva complexitat computacional és elevada, NP-complet, els algoritmes de computació evolutiva són una bona solució per guiar el procés de cerca.

- **Modelatge *(Es desconeix el model)*:** En aquest cas es tenen dades de les entrades i les seves respectives sortides. L'objectiu d'aquests tipus de problemes és trobar un el model adequat que relacioni les entrades amb les sortides. Aquests tipus de problemes són habituals de la mineria de dades. Un exemple seria el sistema de rating de les asseguradores: Es coneixen dades sobre els assegurats (sexe, edat, estat civil, treball...) i també es coneix la taxa de sinistralitat de cada un d'ells, utilitzant algoritmes de computació evolutiva [24] es pot intentar esbrinar el model que classifiqui el client segons el risc depenguen de les seves dades.

- **Simulació *(Es desconeix la sortida)*:** Coneguen l'entrada i el model, els problemes de simulació es poden utilitzar per resoldre problemes del tipus *'que passaria si'* i són útils a l'hora de comprovar els efectes que determinats esdeveniments tindrien

[5]Els problemes que s'utilitzaran en aquest PFC són d'aquest tipus

donat un model conegut, per exemple: es vol conèixer els efectes que determinades decisions empresarials tindrien sobre el flux de caixa.

Un bon exemple, que servirà per exposar el potencial d'aquest tipus d'algoritmes, és l'ús que va fer Keane [33] per a dissenyar el braç de l'antena d'un satèl·lit:

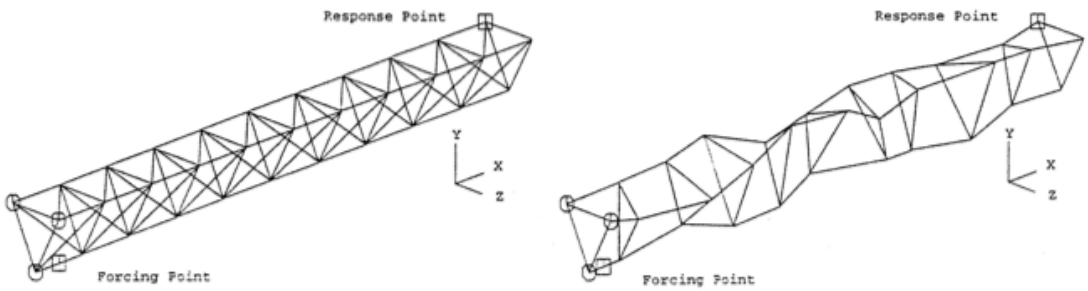
Quan és construeixen determinats tipus de estructures és important reduir el soroll i les vibracions mecàniques ja que aquestes, poden tenir efectes negatius en el funcionament del sistema. Aquest factor és més importat si cap quan estam parlant d'estructures tan cares i complexes com les utilitzades en l'industria aeroespacial.

Per tal de reduir aquest soroll i vibracions hi ha diversos sistemes:

- Ús de materials amb capacitat per absorbir vibracions.
- Sistemes anti-vibració actius.
- Canvis en la geometria de l'estructura

És aquest darrer punt, els canvis en la geometria, el que Keane vol utilitzar per tal de reduir les vibracions i soroll d'un braç metàl·lic que connecta un satèl·lit amb la seva antena.

Per tal d'aconseguir-ho, parteix d'un disseny existent, figura 2.4a, i utilitzant un GA intenta optimitzar[6] la geometria de l'estructura. El resultat obtingut, figura 2.4b, tot i que contra-intuïtiu, es un 20.000% més eficient que l'estructura original.



(a) Disseny original **(b)** Disseny generat pel GA

Figura 2.4: Braç del satèl·lit

L'avantatge de la nova estructura radica en la seva asimetria, ja que aquesta fa que sigui més probable que dues ones vibratòries es trobin en els nodes de l'estructura en diferent fase i per tant, s'anul·lin.

Aquest és un bon exemple que demostra com usant operacions molt simples, com són les operacions genètiques que utilitzen els GA, per analitzar un problema, es pot arribar a una solució òptima.

[6]Coneixem el model i la sortida desitjada

Capítol 3

Descripció i fonaments dels algoritmes utilitzats

En aquest capítol s'exposaran les principals decisions de disseny dels diferents algoritmes així com la seva motivació. L'única decisió de disseny que afectarà a tots els algoritmes per igual és el tamany de la població μ, el qual seguint l'exemple de Karaboga [32] serà de 50 individus[1].

3.1 Algoritmes genètics (GA)

3.1.1 Fonaments

Com ja s'ha explicat, secció 2.2, Holland va ser el primer en introduí aquest tipus d'algoritmes [30], i el seu funcionament teòric està basat en el que s'anomena *schema theorem* [20, pag. 190] [19, pag. 159] [42, pag. 128].

Un **schema** és una representació de la solució candidata allà on determinades posicions estan sense especificà. Per exemple, si la solució candidata està representada per una cadena binaria de 7 bits, un possible *schema* seria: 11****0 on tenim tres bits definits i 4 no definits. Una solució que pertany a aquest esquema s'anomena **instancia**, e.g. **11**0100**0**. A més, el nombre de possibles instàncies de un determinat esquema esta determinat per a^l, on a representa la grossària de l'alfabet utilitzat per a representar la solució i l representa el nombre de posicions sense determinar que hi ha a l'esquema, en el cas de l'exemple donat abans es podrien generar 16 possibles instàncies (2^4).

La teoria de *schema* desenvolupada per Holland prediu el nombre de vegades que un especific esquema apareixerà a la següent generació d'un GA, donat el *fitness* mitjà de la població així com altres factors com el rati de mutació [19]. A més diu que si el *fitness* mitja de les instàncies d'un *schema* està per sobre la mitja, el nombre de instàncies de l'esquema augmentarà dins la població amb el pas del temps [42, pag. 162].

[1] Sol·lucions candidates

És important remarcar, que l'afecte d'aquesta teoria només s'aplica al que s'anomena Simple Genetic Algorithm (SGA), un conjunt molt simple de GA. Aquest tipus de algoritmes es caracteritzen per utilitzar la següent configuració [20, pag. 38]:

- **Representació *genòmica*:** Binaria
- **Combinació:** Intercanvi sobre un punt
- **Mutacio:** Inversió de bit
- **Selecció paterna:** Proporcional al *fitness*
- **Selecció dels supervivents:** Generacional

Als següents apartats s'explicarà amb més detall la configuració utilitzada en aquest PFC i les diferències amb els SGA. De totes maneres, s'ha de tenir en compte que tot i que els la definició de cada component canvií, l'esquema de funcionament de l'algoritme és igual:

```
begin
  S'inicia la població
  while ¬Condició de fi do
        Avaluació de les solucions candidates que formen la població
        Selecció dels pares que generaran la nova generació
        Encreuament dels genotips dels pares per generar un nou individu
        Procés de mutació per els nous individus
        Selecció dels individus que formaran la nova població
  od
end
```

3.1.2 Tipus de representació utilitzat

La representació binaria utilitzada per els SGA té una serie d'avantatges a l'hora de realitzar el procés de mutació i combinació, com veurem més endavant, però donat que el domini de les funcions d'avaluació utilitzades són els nombres reals s'ha considerat més oportú per a la realització d'aquest PFC l'utilització d'una representació utilitzant aquest tipus de valors. El *cromosoma* de cada individu serà un vector de d nombres reals, on d és el nombre de dimensions a optimitzar per a cada funció.

3.1.3 Metodologia de selecció dels 'pares'

Tradicionalment, els SGA utilitzen un mètode anomenat Fitness Proportional Selection (FPS) per assignar a cada solució candidata una probabilitat de ser seleccionat com a *pare* d'un nou individu. Aquest genera el valor a partir de la següent equació:

$$p = \frac{\text{fitness de l'individu}}{\text{Fitness mitjà de la població}}$$

Segons Eiben i Smith [20, pag. 59] aquest sistema té diversos inconvenients:

- **Convergència Prematura:** Individus que són molt millor que la resta col·lapsen la diversitat de la població.
- **Poca pressió de selecció:** Si pel cas contrari tots els individus tenen *fitness* similars, llavors la selecció dels pares seria quasi uniformement aleatòria. En aquest cas, un individu que tingui una petita millora sobre un altre no veu incrementades de manera significativa les seves opcions de paternitat i per tant, aquesta millora tardara en replicar-se per la població.

Tinguen en compte aquests factors, J.E. Baker proposa un mètode d'assignació per rànquing [9]. El mètode d'assignació de probabilitat per a un determinat rànquing és arbitrari mentre la suma de probabilitats de la població sigui unitària, i per tant, pot ser duit a terme de diverses maneres. En aquest projecte s'utilitzara una assignació linealment decreixent tal com exposa Eiben [20].

Aquest mètode d'assignació està parametritzat per s [2]. Si la pròxima generació té la mateixa mida, μ, que l'actual, s determina el nombre esperat de descendents que tindrà l'individu més apte. Si definim el rànquing i com l'interval entre el pitjor individu, rang 0, i el millor, rang $\mu - 1$ llavors la probabilitat de la selecció de i és:

$$P(i) = \frac{2-s}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

El valor de s utilitzat en aquest experiment serà **1.5**, un valor mitja donat l'ampli rang de funcions a optimitzar.

Implementació

El problema tant del mètode de rànquing com de FPS és que els individus no poden ser representats amb la exacte proporció calculada ja que el tamany de la població és finit. S'ha de trobar un mètode per tal de poder seleccionar els futurs *pares* realitzant un mostreig utilitzant la probabilitat de selecció.

Un dels mètodes més coneguts és el de la ruleta. De manera conceptual és l'equivalent a realitzar μ jocs a una ruleta dividida amb μ porcions, cada una amb una àrea determinada per la probabilitat de reproducció[3]. La porció, la que tingui la marca (veure figura 3.1a), que surti guanyadora a cada joc passa al conjunt de pares[4].

El problema, és que s'ha demostrat que aquest mètode no és una bona eina de mostreig ja que tot i que té biaix zero, no garanteix una mínima dispersió. James Baker introdueix el que anomena com Stochastic Universal Sampling (SUS) [9] per tal de corregir els defectes de l'algoritme de la ruleta. SUS a nivell conceptual funciona amb la

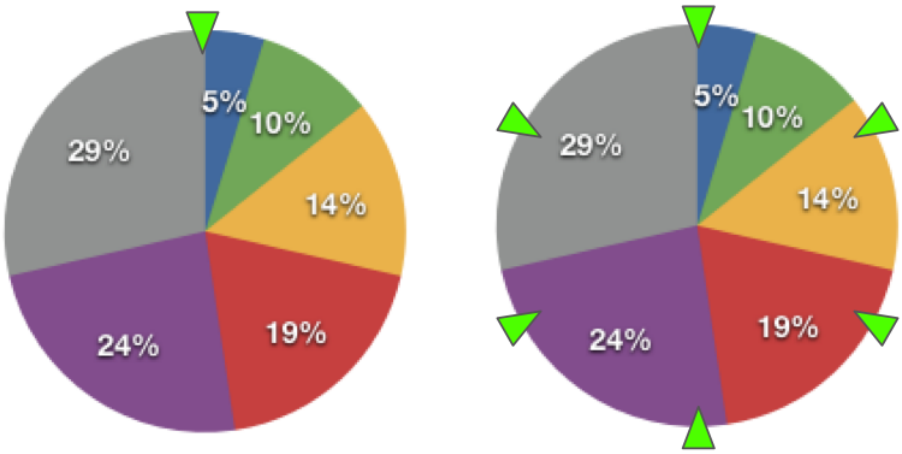
[2] $1.0 < s \leq 2.0$

[3] La suma de les probabilitats a de estar normalitzada

[4] Mating pool

mateixa ruleta abans descrita però aquest cop només es realitza un llançament i els *pares* queden determinats per les μ marques equidistants, figura 3.1b. A continuació s'exposa l'algoritme en pseudocodi:

```
begin
  current_member = i = 0;
  r = Valor aleatori uniformement elegit del rang [0,1/μ];
  while current_m ≤ μ do
      while r ≤ a[i] do
          matting_pool[current_memeber] = parents[i];
          r = r + 1/μ;
          current_member = current_member + 1;
      od
      i = i + 1;
  od
end
```



(a) Algoritme ruleta **(b)** Stochastic universal sampling

Figura 3.1: Mètodes de mostreig

3.1.4 *Crossover* i generació del *offspring*

Un cop generada la *mating pool* es pot procedir a generar la nova generació d'individus. Els SGA, el quals codifiquen el genotip de manera binaria, utilitzen un mètode anomenat *one-point crossover*[5] el qual s'implementa de la següent manera:

- Es seleccionen 2 individus diferents[6] de la **mating pool** i s'eliminen d'aquesta.
- Es selecciona de manera uniformement aleatòria 1 punt d'encreuament del rang $[0, nombre\_total\_bits\_cromosoma)$

[5]Especialització del mètode n-point crossover

[6]Aquest punt és important ja que si fossin iguals l'encreuament no generaria variabilitat genètica.

- Es generen dos nous individus intercanviant les cadenes dels 2 *pares* prèviament seleccionats per el punt d'encreuament determinat, figura 3.2

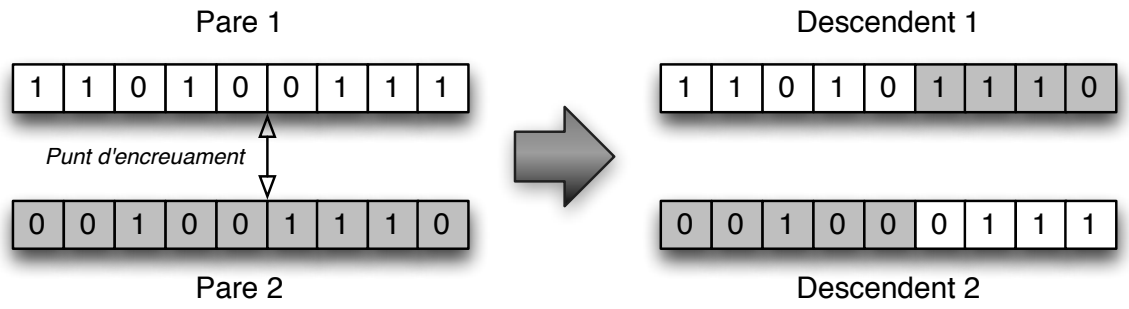


Figura 3.2: Exemple de l'encreuament per un punt

Aquest mètode té l'avantatge que, a part de combinar la informació de dos individus, genera nova informació genètica, ja que el punt d'encreuament pot estar situat el mig de la codificació genètica d'una variable.

En el cas que es vulgui implementar aquest sistema per ser utilitzat en una representació genètica en coma flotant, no tindrem aquesta avantatge ja que no hi ha punts intermedis entre les variables, i per tant l'únic generador del nous valors serà l'operador de mutació.

Per tal d'evitar aquest afecte, Haupt especifica el següent mètode aritmètic [28, pag. 58]:

- Es seleccionen 2 individus diferents de la **mating pool** i s'eliminen d'aquesta.
- Es selecciona la posició n d'una variable de manera uniformement aleatòria
- Es selecciona, també de manera uniformement aleatòria, una variable β continguda en el rang [0,1].
- Sent p_{pare1} el valor de la variable a la posició n pel primer progenitor i p_{pare2} el valor de la variable pel segon. Es generen dos nous valors utilitzant les equacions: $p_{nou1} = p_{pare1} - \beta(p_{pare1} - p_{pare2})$ i $p_{nou2} = p_{pare2} + \beta(p_{pare1} - p_{pare2})$
- Es generen dos nous individus encreuant els cromosomes paterns per la dimensió n. El valor de la dimensió n queda definit per p_{nou1} i p_{nou2} per a cada nou individu respectivament.

3.1.5 Procés de mutació

En el cas dels SGA, cada bit que forma el genotip té una probabilitat p_m d'invertir el seu estat. Això és coneix com a mutació per inversió, *bit flip*.

El procés per a un GA que utilitzi un representació en coma flotant és similar. Cada component real del conjunt de solucions candidates té una probabilitat p_m de sofrir una mutació. El component *mutant* agafarà un valor dins el rang de cerca determinat. L'elecció del valor tradicionalment és fa utilitzant un d'aquests sistemes [20, pag. 44]:

- **Uniformement:** Es selecciona un valor de manera uniformement aleatòria inclòs dins el rang de cerca.
- **Fixada per una distribució:** La probabilitat d'elecció queda delimitada per la distribució seleccionada. Per exemple, en el cas d'utilitzar una distribució normal, es pot definir el valor mutant com $c_{nou} = c_{antic} + \sigma N_n(0,1)$, on c representa el component i σ representa la desviació estàndard de la distribució.

En el cas d'aquest experiment s'ha escollit la primera opció ja que segons Haupt [28, pag. 62] la segona opció té uns quants inconvenients: la seva eficiència està condicionada a la correcte elecció del valor σ. L'adició d'un nombre aleatori pot causar que la variable excedeixi els seus límits. A més, aquest procediment consumeix més temps que el primer.

El valor de $\mathbf{p_m}$ per a la realització del PFC és de **0.02**. Ha estat seleccionat després de realitzar diverses proves tinguen en compte factors com la diversitat de la població o la rapides de convergència cap a l'òptim.

3.1.6 Selecció dels individus que formaran la nova generació

Sigui μ l'actual generació i λ els descendents d'aquesta. En els SGA la nova generació estarà formada únicament pels descendents de la generació actual, $\mu_{nova} = \lambda$. Això és coneix com a reemplaçament generacional.

Aquest sistema té un inconvenient principal, si entre els progenitors hi ha membres amb un *fitness* més alt que qualsevol dels membres de la nova generació, l'informació d'aquests es perdrà quan es crei la nova generació. Un mètode per resoldre aquesta problemàtica és seleccionar els n millors individus de la població actual i passar-los a la nova generació. Concretament:

- Es seleccionen els n millors membres de la població de *pares*, μ. n està determinat pel **generation gap**, percentatge de la població original a ser substituït, en aquest cas el valor assignat és **0.9**, i per tant el valor de n és: $n = \text{nombre d'individus} \cdot (1 - \text{generation gap}) = 50 \cdot 0.1 = 5$
- S'ajunten aquests millors individus amb els individus de la nova generació.
- Es seleccionen els μ millors del conjunt, ja que la mida de la població s'ha de mantenir constant el llarg del pas de les generacions.

3.2 Evolució diferencial (DE)

3.2.1 Fonaments

Storn i Price proposaren aquests sistema com una millora respecte els mètodes de cerca existents[7] ja que aquest convergeix més ràpidament cap a l'òptim, utilitza menys paràmetres de configuració i a més no requereix que la funció a optimitzar sigui derivable [44].

Aquests mètodes s'aplica a funcions reals, i fonamentalment queda descrit per el següent algoritme:

```
begin
  Iniciam la població
  while ¬condicions_fi do
        for i = 0 to len(poblacio) step 1 do
            individu = poblacio[i]
            Cream un vector mutant en base a 3 membres de la població
            Encreuam aquest vector amb l'individu i cream un descendent
            Avaluam l'individu i el descendent
            Afegim a la futura generació el que tengui una millor fitness
        od
  od
end
```

Com es pot veure el principal generador de diversitat és l'operador de mutació que a més funciona en base a vectors existents, amb la conseqüent millora en el procés de cerca. A continuació s'exposaran en més detalls aquests operadors.

3.2.2 Tipus de representació utilitzat

Com en el cas dels algorimtes genètics, secció 3.1.2, s'utilitzarà un vector compost per d components, cada un dels quals representa una dimensió de la funció a optimitzar.

3.2.3 Procés de mutació

Com s'ha dit abans, el principal generador de diversitat és l'operador de mutació. Per a cada un dels vectors presents, v_i, a la població es crea un vector mutant basat en uns altres tres membres seleccionats aleatòriament, x_{r_1}, x_{r_2}, x_{r_3} tots ells diferents[8]. El vector mutant, v_m, es genera en funció a:

$$v_m = x_{r_1} + F(x_{r_2} + x_{r_3})$$

El paràmetre F s'utilitza com a factor d'escalat i està definit dins el rang [0,1]. En aquest experiment s'utilitza el valor **0.4717** tal com recomana Pederesen [40].

[7] Existents a l'any de la publicació del mètode, 1995

[8] $x_{r_1} \neq x_{r_1} \neq x_{r_2} \neq x_{r_3} \neq v_i$

3.2.4 Mètode de *crossover*

Un cop s'ha creat aquest vector mutant, v_m, arriba l'hora de realitzar l'encreuament. En els DE l'encreuament entre el v_m i v_i és realitza dimensió per dimensió i el descendent queda definit per:

$$v_{desc}^d = \begin{cases} v_m^d & R_d \leq CR \\ v_i^d & R_d > CR \end{cases}$$

En aquest cas es genera de manera uniformement aleatòria un valor dins l'interval [0,1] per a cada dimensió, R_d, i és compara amb un valor constant prèviament seleccionat CR, en aquest cas **0.8803** [40]. Si el valor R_d és inferior o igual al valor de CR la dimensió d del nou individu serà la del vector mutant en cas contrari serà la de v_i.

3.2.5 Selecció de la propera generació

Un cop finalitzat aquest procés és compara v_i amb el seu descendent i es selecciona per a formar part de la nova generació el que tingui una millor aptitud.

3.3 Optimització amb eixam de partícules (PSO)

3.3.1 Fonaments

Els fonaments dels PSO es basen en la teoria sociocognitiva on el procés d'adaptació cultural està dividit entre els components d'alt nivell, formació de patrons entre individus i la habilitat de resoldre problemes, i els de baix nivell inherents al comportament de cada individu els quals poden ser resumits en [34]:

- **Avaluació:** Capacitat d'avaluar els components de l'entorn i reaccionar positiva o negativament a aquests.
- **Comparació:** Comparació de l'estat propi amb l'estat dels individus pròxims.
- **Imitació:** Si a través de la comparació és descobreixen qualitats desitjades per l'individu, aquest tendirà a imitar el comportament de l'individu que les hagi aconseguit.

En base a aquestes observacions, Kennedy i Eberhart proposen un nou model per a l'optimització de problemes [35] el qual defineix partícules, i.e. solucions candidates, que es van moguen per l'espai de possibles solucions. El funcionament bàsic seria:

```
begin
  Generam una població inicial
  Definim el veïns per a cada individu
  while ¬condicions_fi do
        for i = 0 to len(poblacio) step 1 do
            individu = poblacio[i]
            Desplaçam l'individu tinguen en compte el seu millor fitness passat
            i el millor fitness d'entre els seus veïns
        od
  od
end
```

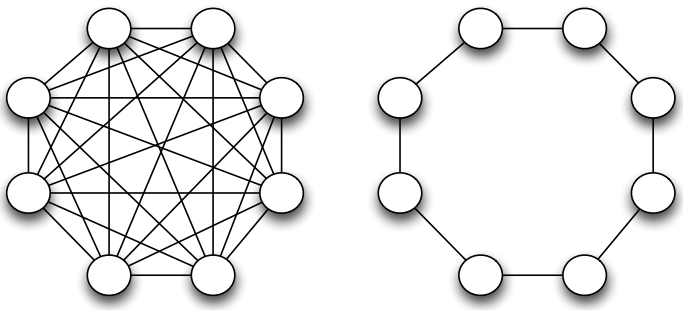
3.3.2 Topologia

El factor de la topologia està relacionat l'estructura social de les partícules, amb com aquestes interactuen, comparen i influencien a altres partícules veïnes. S'ha de remarcar que aquestes partícules veïnes no tenen perquè ser pròximes dins l'espai de solucions:

> [...]neighbors are not necessarly individuals who are near them in the parameter space, but rather ones that are near them in topological space that defines the sociometric structure of the population. Who influences you is defined by your connections in the social network, not by positions in a belief space [19, pag. 311]

Hi ha diversos tipus de topologia depenguen de les necessitats de l'optimització, però les més habituals són:

- **Connexió total:** Tots els individus estan connectats amb tots, figura 3.3a.
- **Connexió amb anell:** Cada individu està connectat amb el seu predecessor i el seu antecessor, figura 3.3b.



(a) Connexió total **(b)** Connexió amb anell

Figura 3.3: Topologies

L'elecció d'una o altre, com s'ha comentat, depèn de les necessitats. Una connexió total permet una convergència més ràpida a costa d'augmentar la possibilitat de convergir en un mínim local. Passa el contrari en cas que s'implementi una topologia amb anell.

En el cas de l'experiment a realitzar, la prioritat és l'eficàcia, i per tant una convergència ràpida a un mínim no és necessària ja que el nombre màxim d'iteracions permeses és elevat. Per tant la topologia utilitzada serà una **topologia amb anell**.

3.3.3 Representació dels paràmetres

Ara que ja s'ha definit el concepte de topologia, es pot donar la representació utilitzada per a cada individu. Aquesta estarà composta per:

- La posició actual de la partícula. Representada per un vector amb d dimensions, $x_i(t)$
- La darrera velocitat utilitzada per calcular el desplaçament de la partícula, $v_i(t)$
- La millor posició[9] aconseguida per la partícula durant tota l'execució, x_{lbest}
- Referència a 2 veïns.
- La millor posició aconseguida fins el moment d'entre les millors posicions del conjunt de veins[10], x_{nbest}

3.3.4 Càlcul del desplaçament

La nova posició es calcula a partir de les millors posicions conegudes (pròpia i del conjunt de veïns), la darrera posició ocupada per la partícula i la velocitat, vector de desplaçament, d'aquesta. En concret per a cada partícula de la població es realitzen els següents càlculs:

for $d = 1$ to $nombre\_de\_dimensions$ step 1 do
$\quad v_i^d(t+1) = v_i^d(t) + \varphi_1(x_{lbest}^d - x_i^d(t)) + \varphi_2(x_{nbest}^d - x_i^d(t))$
$\quad x_i^d(t+1) = x_i^d(t) + v^d(t+1)$
od

El valor de φ representa el pes que cada una de les millors posicions conegudes té sobre la partícula a l'hora de calcular el seu moviment. Aquest valor es selecciona de manera uniformement aleatòria en l'interval que va des de zero fins a un limit estipulat. En aquest experiment tant φ_1 com φ_2 tenen el mateix limit *2.05* tal com indica Clerc [14].

[9] S'entén que una posició és millor que una altre si aquesta té millor *fitness*
[10] Inclosa la pròpia partícula

3.3.5 Velocitat màxima

El procediment esmentat a l'apartat anterior té un inconvenient, no controla la velocitat a que la partícula pot arribar a accelerar-se. I aquest és un problema important ja que si la partícula continua accelerant-se de manera indefinida no convergirà a la zona del mínim. La manera tradicional afrontar aquest inconvenient és establit un rang de velocitats permeses delimitat per v_{max}:

$$v_i^d(t+1) \in (-v_{max}, v_{max})$$

L'elecció de un bon valor v_{max} depèn de múltiples factors: rang de cerca, nombre de partícules, funció d'avaluació... Això pot suposar un problema quan l'algoritme a de fer front a diversos tipus de funcions. Per aquest motiu, en aquest experiment no s'utilitzarà aquesta metodologia, i optarem per *limitar* la velocitat utilitzant el mètode anomenat *constricció simple.*

3.3.6 Constricció simple (χ)

Quan el valor de φ utilitzat és major que 4, com es el cas[11], es pot utilitzar el mètode anomenat de constricció simple [14], el qual consisteix a aplicar un factor χ al càlcul de la velocitat:

$$v_i^d(t+1) = \chi(v_i^d(t) + \varphi_1(x_{lbest}^d - x_i^d(t)) + \varphi_2(x_{nbest}^d - x_i^d(t)))$$

El valor de χ és calcula en base:

$$\chi = \frac{2k}{\left|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}\right|}$$

On la variable k pren un valor entre zero i u. En el cas de l'experiment k pren el valor 1 tal com suggereix Kennedy [19, pag. 339], i per tant el valor de χ serà aproximadament **0.73**.

La constricció simple té l'avantatge de permetre passar del mode d'exploració[12] al mode d'explotació[13] i viceversa depenguen de l'espai explorat fins un determinat moment.

3.4 Colònia artificial d'abelles (ABC)

3.4.1 Fonaments

Les abelles de la mel *(Apis mellifera)* formen estructures socials complexes (Eusocialitat) on cada abella té un paper important. A dins els algoritmes d'intel·ligència en eixam aquest factor és important ja que, a part de tenir un sistema auto-organitzat sorgit de

[11] $\varphi = \varphi_1 + \varphi_2 = 2.05 + 2.05 = 4.1$
[12] Cerca de mínims
[13] Millora dels resultats per un determinat mínim

l'interactuació dels components de baix nivell(secció 3.3.1), la divisió de treball amb individus especialitats afavoreix l'adaptació davant els canvis en les condicions del medi [31].

El 2005 Darvis Karaboga va introduir un algoritme(ABC) [31] basat en el treball que Tereshko havia fet modelant el comportament de l'abella de la mel a l'hora de compartir informació sobre les fonts d'aliment [45]. Aquest model consisteix en tres components:

- **Font d'aliment:** l'elecció d'una font o una altre depèn de la seva valoració basada en factors com proximitat, qualitat del nèctar, facilitat d'extracció. A l'hora de parlar dels ABC aquest component fa referència a una solució candidata i la seva valoració esta basada en la funció d'avaluació.

- **Abelles recol·lectores ocupades:** Abelles que actualment estan explotant una font d'aliment. Aquesta abella posseeix informació sobre aquesta i la transmet a les altres abelles quan va al rusc a descarregar el nèctar.

- **Abelles recol·lectores no ocupades:** Aquestes abelles no tenen assignada cap font d'aliment i per tal de trobar-ne alguna poden:

  - Cercar fonts d'aliments aleatòriament per l'entorn (Entre un 5% i un 10% del total d'abelles desocupades ocupen aquest rol i són anomenades escoltes).
  - O bé escollir una font d'aliment basant-se en l'informació que les abelles ocupades transmeten quan arriben al rusc. Aquests tipus d'abelles s'anomenen reclutes.

L'intercanvi d'informació entre les abelles ocupades i no ocupades té lloc a una zona comú a la majoria de ruscs, l'àrea de ball. En aquesta les abelles desocupades esperen que les abelles ocupades arribin i compartesquin informació sobre les seves respectives fonts d'aliment. Les desocupades es divideixen entre les diferents zones d'aliment proporcionalment a la valoració d'aquestes, passant de desocupades a ocupades[14].

Per exemplificar aquest funcionament, Karaboga proposa el diagrama de la figura 3.4 [31] on EF representa a les abelles ocupades, UF les desocupades, S les escoltes i R les reclutes.

Ara que ja s'ha introduït la divisió de treball entre els diferents tipus d'abella, es pot especificar quins són, en aquest cas, els components de baix nivell de l'estructura social:

- **Avaluacions positives:** Així com la qualitat de una font d'aliment augmenta, el nombre d'abelles que l'*exploten* també augmenta.

- **Avaluacions negatives:** Si no hi ha millora s'abandona la font d'aliment.

- **Fluctuacions:** Les abelles escolta duen a terme el procés d'exploració de l'entorn cercant aliment en punts aleatoris.

[14] Aquest tipus de abella normalment és coneix com a recluta

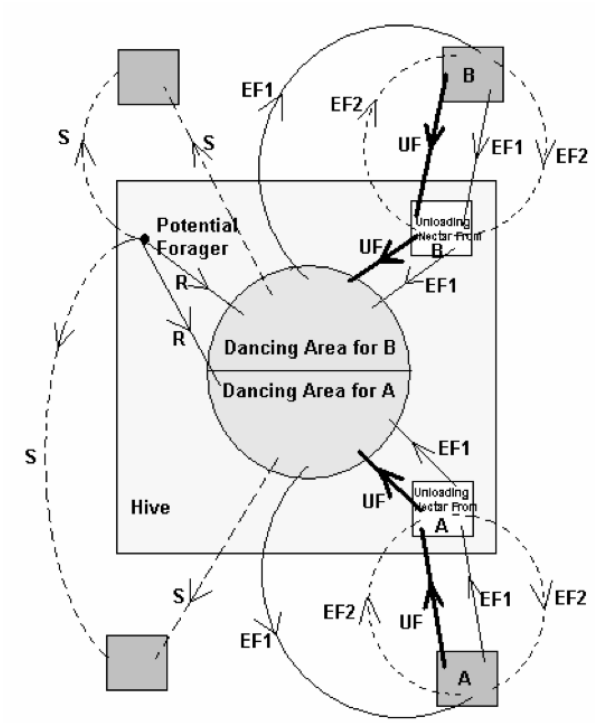


Figura 3.4: Comportament de les abelles de la mel

- **Múltiples interaccions:** Les abelles comparteixen informació amb els seus congèneres al rusc.

El funcionament bàsic de l'algoritme és el que es pot veure en a continuació:

begin
  Iniciam la població de possibles solucions
  while $\neg condicions\_fi$ do
    Posicionar les abelles ocupades a cada una de les possibles solucions
    Distribuir les abelles no ocupades entre les solucions proporcionalment a la seva valoració
    Enviar a les escoltes a cercar noves fonts d'aliment
    Memoritzar la millor posició trobada fins el moment
  od
end

3.4.2 Funcionament de l'algoritme

Una de les avantatges que aquests algoritmes tenen sobre la resta dels presentats, es el seu reduït nombre de paràmetres de control. En concret:

- Nombre de fonts d'aliment, solucions candidates, possibles *SN*. Aquest valor es manté constant durant tota l'execució, 50.

- Nombre màxim d'iteracions permeses per una determinada font d'aliment sense que el procés d'explotació hagi millorat el valor de la seva *fitness*, *limit*. Aquest valor normalment està determinat pel nombre de dimensions de cada solució candidata i el nombre de solucions candidates: $limit = SN \cdot D$ tal com indica Karaboga [32].

A continuació desglossarem els diferents passos que per a cada iteració composen l'algoritme:

1. Cada possible àrea d'aliment té assignada una abella, que l'explota modificant una dimensió j de la seva posició actual, seleccionada de manera uniformement aleatòria en base a: $v_i^j = x_i^j + \phi(x_i^j - x_k^j)$ on x_i és la seva posició actual, x_k és la posició d'una altre abella situada a una àrea d'aliment diferent seleccionada de manera aleatòria i *phi* és un nombre aleatori escollit dins l'interval [-1,1]. És important que x_i i x_j siguin diferents.

   (a) Si la valoració de la nova solució candidata és menor a la prèvia, es descarta i s'augmenta de manera unitària el comptador del nombre de vegades que s'ha explorat l'àrea sense trobar millores.

   (b) En cas contrari, s'adopta la nova posició com la millor de l'àrea i es posa el comptador a zero.

2. Es calcula la probabilitat que una abella no ocupada sigui enviada a una determinada posició en base a la probabilitat: $p_i = \frac{fit_i}{\sum_{n=1}^{SN} fit_n}$. En base a aquesta probabilitat es distribueixen les *SN* abelles entre les diferents àrees utilitzant l'algoritme SUS exposat a la secció dels GA

3. Es repeteix el pas 1 per a cada abella *recluta* recent assignada.

4. Es guarda el millor resultat aconseguit fins el moment.

5. Si hi ha alguna àrea que hagi arribat al limit de nombre d'exploracions, s'elimina del conjunt de possibles solucions i s'en genera aleatòriament una de nova.

La funció *fit* utilitzada per a calcular la probabilitat representa el *fitness* de la solució. Sorgeix un problema quan el que es pretén és minimitzar una funció, ja que com més petits siguin els valor retornats per la funció d'avaluació *f(x)* millor *fitness* té. Per solucionar-ho s'utilitza la funció intermèdia següent: $fit_i = \frac{1}{1+f(x_i)}$

Capítol 4

Descripció de l'experiment

Com ja es va explicar al capítol introductori, l'experiment consistirà en compara la eficàcia dels ABC *versus* la dels GA, DE i PSO. A continuació es descriurà el procediment que es seguirà per a la seva realització.

4.1 Metodologia

4.1.1 Mesures

Eiben i Smith cataloguen [20, pag. 245] les diferents mesures del rendiment dels algoritmes basats en computació evolutiva dins els següents subconjunts:

- Rati de succés (Nombre de execucions concloses amb èxit/nombre total d'execucions)
- Eficàcia (qualitat de la solució)
- Eficiència (velocitat)

Com ja s'ha esmentat, en aquest PFC s'utilitzarà l'eficàcia a l'hora de realitzar les diferents comparatives. En concret s'utilitzarà el que s'anomena Mean Best Fitness (MBF), mitjana de les millors aptituds.

S'ha de pensar que el comportament d'aquests algoritmes és d'origen estocàstic, i per tant les mesures del rendiment són de naturalesa estadística. En concret, MBF calcula, després de realitzar n execucions de cada algoritme, la mitjana[1] de les n millors aptituds[2] de les diferents execucions.

La comparativa es realitzarà comparant els MBF però amb finalitats informatives, també s'exposarà el millor i el pitjor valor d'aptitud per cada algoritme i funció d'avaluació, obtinguts després de n execucions. Amb la mateixa finalitat també s'exposarà la mitjana del temps utilitzat i el nombre mitja d'iteracions realitzades a les execucions que hagin conclòs amb èxit.

[1] Els valors de les mitjanes aniran acompanyats pel valor de la desviació estàndard i l'error estàndard
[2] Obtingudes mitjançant la funció d'avaluació

4.1.2 Condicions de fi d'execució

Ja s'ha comentat (Apartat 2.1.1) que un dels elements bàsics que inclouen els algoritmes de computació evolutiva són les condicions de fi. Això es degut principalment el seu origen estocàstic, ja que no es pot garantir que la solució donada sigui la millor existent però si una bona aproximació. Aquest fet podria ser intranscendent per problemes de complexitat moderada, on la solució es pot trobar a través d'algoritmes deterministes en un temps raonable, però quan es parla de problemes amb alta complexitat aquesta solució no és viable i es fa certa l'expressió de *Voltaire* al cèlebre *Dictionnaire Philosophique*:

> *Le mieux est l'ennemi du bien.*
> El millor és enemic del bo.

En aquest experiment es donaran dues condicions per finalitzar una execució. La primera és que la millor solució trobada estigui per davall d'un determinat llindar. Aquest llindar es pot establir quan es coneix, en determinada mesura, una solució acceptable del problema. En molts problemes reals és complicat fer aquesta estimació, però en el cas d'aquesta pràctica, es treballara amb funcions de les quals es coneix el seu mínim i per tant es podrà determinar. Aquest serà el mínim més un determinat marge d'error. Aquest marge d'error s'ha establert a $1 \cdot 10^{-10}$. Cal remarcar que si un execució finalitza per aquesta condició es considerà que la execució ha finalitzat amb èxit.

La segona condició limita el nombre de temps que es pot estar executant un algoritme. Ja que la complexitat és elevada i no podem garantir que s'arribi a un valor per davall del llindar indicat, aquest limit ens assegura que l'execució finalitzarà en algun moment tornant el millor valor obtingut fins aquest instant. La limitació imposada en aquest cas fa referència el nombre de generacions/iteracions que una execució pot generar com a màxim. En el cas d'aquest experiment s'ha optat per limitar-ho a **50.000** iteracions, un nombre força elevat donat que en el nostre cas prioritzam més l'eficàcia que la eficiència. Una execució que finalitza sota aquesta condició existint la condició abans esmentada (el llindar), serà considerada no satisfactòria.

4.1.3 Funcions a optimitzar

Els problemes que els algoritmes hauran d'optimitzar estan definits per quinze diferents funcions d'avaluació, cada una d'elles amb diferents propietats. Seguint el criteri de Karaboga [32] es catalogaran les diferents funcions depenent de la seva modalitat i la seva separabilitat.

La modalitat defineix quants mínims té la funció. En concret poden ser:

- **Unimodals:** Les funcions d'aquesta categoria tenen un sol mínim global.
- **Multimodals:** Aquesta categoria de funcions es caracteritza per tenir múltiples mínims *locals* i un sol mínim global.

La modalitat ajudarà a estudiar diversos aspectes dels algoritmes com la rapides de convergència cap a l'òptim o la capacitat per continuar la cerca un cop s'ha convergit a un mínim local, aquest és un dels principals problemes amb que han de tractar els algoritmes.

La separabilitat parla de la manera en la que es composa la funció, tenim dues categories:

- **Separable:** Una funció amb p-variables pot ser expressada com la suma de p funcions de una sola variable.
- **No separables:** No poden ser descrites de la forma anteriorment exposada.

A la taula 4.1 es pot veure la llista de les funcions que utilitzades amb les seves característiques: Nom, formulació, dimensions, intervals, mínim global, categoria[3]. Aquests valors han estat donats gràcies a les especificacions de Karaboga [32] i Molga [39].

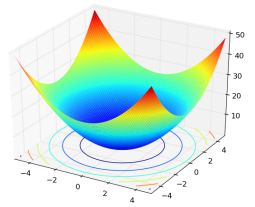
Nom	Formulació	C	D	Int.	Mín
Sphere	$f(x) = \sum_{i=1}^{n} x_i^2$	US	30	[-100,100]	0.0
Step	$f(x) = \sum_{i=1}^{n} (\lfloor x_i + 0.5 \rfloor)^2$	US	30	[-100,100]	0.0
SumSquares	$f(x) = \sum_{i=1}^{n} i x_i^2$	US	30	[-10,10]	0.0
Colville	$f(x) = 100\left(x_1^2 - x_2\right)^2 + (x_1-1)^2 + (x_3-1)^2 + 90\left(x_3^2 - x_4\right)^2 + 10.1\left((x_2-1)^2 + (x_4-1)^2\right) + 19.8(x_2-1)(x_4-1)$	UN	4	[-10, 10]	0.0
Zakharov	$f(x) = \sum_{i=1}^{n} x_i^2 + \left(\sum_{i=1}^{n} 0.5 i x_i\right)^2 + \left(\sum_{i=1}^{n} 0.5 i x_i\right)^4$	UN	10	[-5, 10]	0.0
Schwefel2.22	$f(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \prod_{i=1}^{n} \lvert x_i \rvert$	UN	30	[-10, 10]	0.0
Himmelblau	$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$	MS	2	[-10,10]	0
Rastrigin	$f(x) = \sum_{i=1}^{n} \left[x_1^2 - 10cos(2\pi x_1) + 10\right]$	MS	30	[-5.12, 5.12]	0.0
Schwefel	$f(x) = \sum_{i=1}^{n} -x_i \cdot \sin\left(\sqrt{\lvert x_i \rvert}\right)$	MS	30	[-500, 500]	-12569.5
Michalewicz5	$f(x) = -\sum_{i=1}^{n} \sin(x_i) \cdot \left(\sin\left(\frac{i x_i^2}{\pi}\right)\right)^{20}$	MS	5	[0, π]	-4.6877
Schaffer	$f(x) = 0.5 + \frac{sin^2(\sqrt{x_1^2+x_2^2})-0.5}{(1+0.001(x_1^2+x_2^2))^2}$	MN	2	[-100,100]	0
Shubert	$f(x) = (\sum_{i=1}^{5} icos((i+1)x_1 + i))(\sum_{i=1}^{5} icos((i+1)x_2 + i))$	MN	2	[-10,10]	-186.7309
Shekel10	$f(x) = -\sum_{i=1}^{10} \left(\sum_{j=1}^{4} \left[(x_j - a_{ij})^2 + c_j\right]\right)^{-1}$	MN	4	[0, 10]	-10.5364
Griewank	$f(x) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	MN	30	[-600, 600]	0.0
Ackley	$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n} \cos(2\pi x_i)\right) + 20 + e$	MN	30	[-32, 32]	0.0

Taula 4.1: Funcions d'avaluació utilitzades

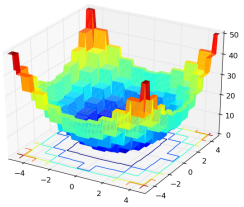
A continuació es presentaran uns quants detalls d'aquestes funcions:

- **Sphere:** També anomenada *primera funció de De Jong* és un de les més simples. És continua, convexa, unimodal i separable. Figura 4.1a
- **Step:** Aquesta és una versió discontinua de la funció de De Jong. Té una forma més escalonada com es pot veure a la representació 3D, figura 4.1b

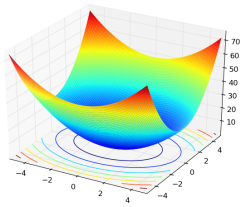
[3]U: Unimodal, M: Multimodal, S: Separable, N: No separable



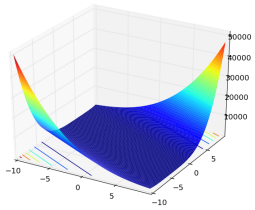
(a) Sphere



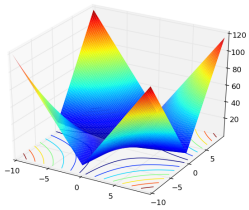
(b) Step



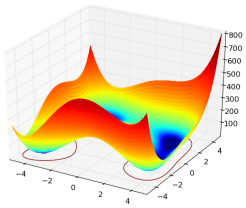
(c) SumSquares



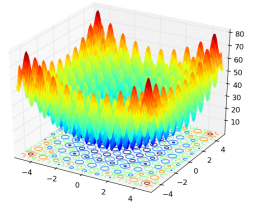
(d) Zakharov



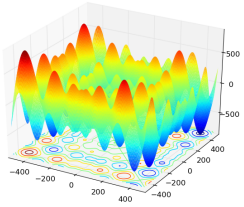
(e) Schwefel 2.22



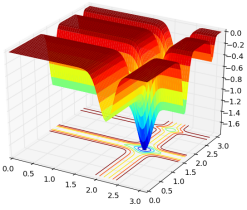
(f) Himmelblau



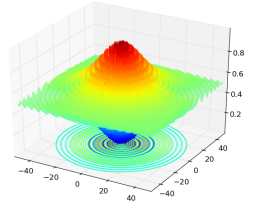
(g) Rastrigin



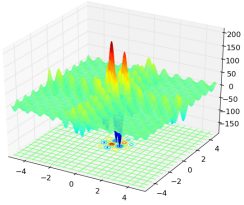
(h) Schwefel



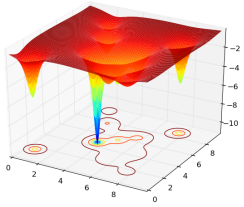
(i) Michalewicz 5



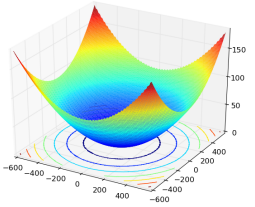
(j) Schaffer



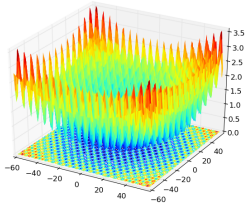
(k) Shubert



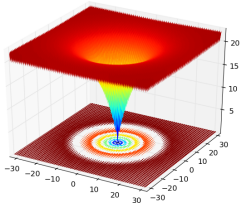
(l) Shekel



(m) Griewank



(n) Zoom Griewank



(o) Ackley

Figura 4.1: Representació 3D de les funcions d'avaluació

- **SumSquares:** La representació és la que es pot veure a la figura 4.1c i també es anomenada *model de l'esfera ponderada*. Es molt similar a la funció de De Jong però té una forma més ovalada.
- **Colville:** D'aquesta funció no és possible donar-ne representació gràfica ja que està composta per quatre variables. Té la dificultat que comparada amb les presentades fins ara aquesta no és separable.
- **Zakharov:** Figura 4.1d, té un mínim poc pronunciat el que pot provocar l'augment en el temps de convergència.
- **Schwefel 2.22:** La funció representada per la figura 4.1e prova la resistència del procés de cerca, ja que valors del *fitness* alts, poden donar problemes a l'hora de determinar la direcció de la cerca [6].
- **Himelblau:** Les ubicacions de tots els mínims es poden trobar analíticament. No obstant això, pel fet que són les arrels dels polinomis cúbics, quan s'escriu en termes dels radicals, les expressions són una mica més complicats d'optimitzar [4]. Figura 4.1f
- **Rastrigin:** Aquesta funció, figura 4.1g, també està basada amb la funció de De Jong, afegint-li una modulació per tal d'obtenir freqüents mínims locals [39, pag. 7]
- **Schwefel:** Funció enganyosa ja que el mínim global és geomètricament distant, en l'espai de paràmetres, dels millors mínims locals. Per tant, els algorisme de cerca són potencialment propensos a la convergència en la direcció equivocada [39, pag. 9] [43]. Representació 3D a la figura 4.1h.
- **Michalewicz5:** A la figura 4.1i es pot veure la representació 3D d'aquesta funció, introduïda per Zbigniew Michalewicz [38]. D'aquesta n'existeixen dues versions, una amb 5 dimensions i una amb 10, i es caracteritza per donar molt poca informació al procés de cerca sobre l'ubicació del mínim global.
- **Schaffer:** Aquesta multimodal té repartits de manera periòdica mínims locals, que és van accentuant així com s'atraquen a l'origen. La figura 4.1j mostra la representació.
- **Shubert:** La complicació d'aquesta funció multimodal rau en el fet que té mínims locals amb un valor similar al del mínim local pròxims a aquest. Figura 4.1k
- **Shekel:** Anomenada també *Shekel's foxholes* per la forma que té la seva representació, figura 4.1l. Està definida per una serie de nombres constants fixats d'antuvi, taula 4.2.
- **Griewank:** L'idea darrera aquesta funció és similar a la ja exposada per a la funció Rastrigin, afegir una modulació a la primera funció de De Jong. Aquest cop, però, utilitzant una funció no separable. A més, l'interval allà on es realitzarà la cerca es considerablement més gran. Es poden veure dues representacions d'aquesta funció a les figures 4.1m i 4.1n. A la primera figura es pot observar la representació de la funció per a l'interval donat, es molt similar a la de De Jong, en canvi si s'acurça l'interval s'observa com aquesta conté nombrosos mínims locals.

- **Ackley:** Funció altament multimodal, figura 4.1o amb nombrosos mínims locals uniformement repartits per l'espai de cerca, però amb un mínim global a l'origen [5] [20, pag. 268].

i/j	**1**	**2**	**3**	**4**	C_i
1	4	4	4	4	0.1
2	1	1	1	1	0.2
3	8	8	8	8	0.2
4	6	6	6	6	0.4
5	3	7	3	7	0.4
6	2	9	2	9	0.6
7	5	5	3	3	0.3
8	8	1	8	1	0.7
9	6	2	6	2	0.5
10	7	3.6	7	3.6	0.5

Taula 4.2: Constants utilitzades per a la funció Shekel

4.1.4 Metodologia utilitzada

Tots els algoritmes utilitzats hauran de *cercar* el mínim de cada una de les funcions presentades. A més, donada la participació de l'atzar en el funcionaments dels algoritmes, haurem de repetir n vegades[4] el procés de *cerca* per cada una de les funcions i algoritmes per tal de poder aplicar eines estadístiques per analitzar-los. En aquest experiment el valor de n es 31 per tal poder treballar amb una distribució normal gràcies al Teorema del Límit Central:

> El **teorema del límit central** diu que si una mostra és prou gran (n>30), sigui quina sigui la distribució de la variable d'interès, la distribució de la mitjana mostral serà aproximadament una normal. A més, la mitjana serà la mateixa que la de la variable d'interès, i la desviació típica de la mitjana mostral serà aproximadament l'error estàndard [8, Mòdul 5, pag. 17].

Un cop es tenguin els estadístics per a cada funció i algoritme es procedirà a realitzà la comparativa. Es compararan independentment els resultats dels ABC per cada una de les funcions amb els resultats obtingut per a les mateixes funcions utilitzant els GA, DE i PSO. El tractar-se del contrast de dues mostres independents s'utilitzarà un t-test amb un nivell de significació α de 0.05.

Donat que hi ha diverses funcions, s'han de realitzar múltiples comparacions per a cada parell d'algoritmes[5]. Tenguen en compte aquest factor, s'ha de tenir present la **desigualtat de Boole**:

[4] Cada vegada iniciant el generador pseudoaleatori amb una llavor diferent
[5] ABC *vs* GA, ABC *vs* DE i ABC *vs* PSO

> A la teoria de la probabilitat, la desigualtat de Boole estipula que per tota família finita o numerable de successos, la probabilitat de que el menys un succeeixi es menor a la suma de les probabilitats dels successos individuals [1].

Per tal d'evitar els falsos positius derivats d'aquesta desigualtat, el matemàtic italià Carlo Emilio Bonferroni introduí la **correcció de Bonferroni**. El seu funcionament és el següent:

- S'ordena de major a menor les comparatives de les diferents funcions depenguen del seu *p-vaule*.
- Com a nivell de significació per a cada funció s'agafa el valor inicial de α dividit per la posició que ocupa dins la llista prèviament ordenada.

S'ha de destacar que aquest mètode té alguns inconvenients com el fet que així com es va reduït la probabilitat de cometre falsos positius s'augmenta la de cometre falsos negatius. Donat que el nombre de múltiples comparacions no és molt elevat, quinze comparatives, no es tindran en compte les implicacions d'aquests falsos negatius.

Capítol 5

Resultats

5.1 Resultats de les execucions

A les següents taules es mostra'n els estadístics de les execucions dels algoritmes a l'hora d'optimitzar les funcions anteriorment exposades, comentant-ne, si escau, els detalls més rellevants.

Funció	Mínim		GA	DE	PSO	ABC
sphere	0	Fitness mean	4.59774e-05	0	0	0
		Fitness stdDev	3.55089e-05	0	0	0
		Fitness SEM	6.483e-06	0	0	0
		Min Fitness	6.08e-06	0	0	0
		Max Fitness	0.000139637	0	0	0
		Success Ratio	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	842.839	1376	1022.19
		Iterations stdDev	-	23.7882	27.1543	36.1988
		Iterations SEM	-	0.0184345	0.0309023	0.0291854
		Min Iterations	-	802	1322	956
		Max Iterations	-	937	1425	1095
		Duration mean	-	4.63776	12.0851	6.05219
		Duration stdDev	-	0.10097	0.169259	0.159855
		Duration SEM	-	0.0184345	0.0309023	0.0291854
		Min Duration	-	4.39563	11.7193	5.70982
		Max Duration	-	4.95507	12.4157	6.43809

Taula 5.1: Resultats per a la funció sphere

Les taules 5.1, 5.2 i 5.3 mostren els estadístics per a les que, a priori, són les funcions més senzilles: unimodals i separables. Els primers resultats a analitzar, els obtinguts de l'optimització de la funció *Sphere*, mostren com els algoritmes DE, PSO i ABC no han tingut cap problema a l'hora de trobar el mínim amb la precisió requerida, a més, la convergència a aquest ha estat bastant ràpida, com és pot veure en el nombre mitja d'iteracions necessàries. L'algoritme GA, en canvi, no han aconseguit després de 50.000 iteracions el seu objectiu, quedant 5 ordres de magnitud per sobre del valor desitjat.

L'optimització de la funció *Step*, taula 5.2, ha finalitzat amb èxit per cada un dels algoritmes utilitzats, fet motivat principalment per la major àrea ocupada pel mínim de la funció. Sobre la velocitat de convergència, pareix que tan ABC com DE han necessitat significativament menys iteracions per convergir que els altres.

Funció	Mínim		GA	DE	PSO	ABC
step	0	Fitness mean	0	0	0	0
		Fitness stdDev	0	0	0	0
		Fitness SEM	0	0	0	0
		Min Fitness	0	0	0	0
		Max Fitness	0	0	0	0
		Success Ratio	100.00%	100.00%	100.00%	100.00%
		Iterations mean	1477.03	276.129	1558.39	118.226
		Iterations stdDev	296.084	19.7137	2328.99	15.8759
		Iterations SEM	0.11717	0.0148768	2.08011	0.0161338
		Min Iterations	1038	250	503	89
		Max Iterations	2238	351	12866	158
		Duration mean	4.79786	1.71311	10.7316	0.810739
		Duration stdDev	0.641766	0.0814836	11.3932	0.0883683
		Duration SEM	0.11717	0.0148768	2.08011	0.0161338
		Min Duration	3.62985	1.58147	4.82516	0.652617
		Max Duration	6.3967	1.95471	65.9131	1.04514

Taula 5.2: Resultats per a la funció step

Funció	Mínim		GA	DE	PSO	ABC
sumSquares	0	Fitness mean	6.915e-06	0	0	0
		Fitness stdDev	3.441e-06	0	0	0
		Fitness SEM	6.282e-07	0	0	0
		Min Fitness	2.1179e-06	0	0	0
		Max Fitness	1.64886e-05	0	0	0
		Success Ratio	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	783.839	1292.03	930.29
		Iterations stdDev	-	35.6742	29.6185	30.3615
		Iterations SEM	-	0.0272109	0.0384227	0.0329522
		Min Iterations	-	729	1221	864
		Max Iterations	-	940	1360	994
		Duration mean	-	4.62715	11.9036	6.32655
		Duration stdDev	-	0.14904	0.21045	0.180487
		Duration SEM	-	0.0272109	0.0384227	0.0329522
		Min Duration	-	4.29739	11.3543	5.87629
		Max Duration	-	5.12905	12.2759	6.67836

Taula 5.3: Resultats per a la funció sumsquares

L'última funció d'aquesta categoria, la funció *SumSquares*, ha resultat ser molt similar a la *Sphere* en quan els estadístics resultats de la seva optimització. Els ABC, DE i PSO han conclòs amb èxit totes les execucions i en canvi els GA han quedat 4 ordres de magnitud per sobre del valor requerit.

Funció	Mínim		GA	DE	PSO	ABC
colville	0	Fitness mean	0.39419	0.0827249	0	0.0159363
		Fitness stdDev	0.391067	0.23566	0	0.00735529
		Fitness SEM	0.0713987	0.0430254	0	0.00134289
		Min Fitness	0.00137435	0	0	0.00318176
		Max Fitness	1.12703	0.900947	0	0.0321819
		Success Ratio	0.00%	83.87%	100.00%	0.00%
		Iterations mean	-	193.731	9374.35	-
		Iterations stdDev	-	21.5541	943.285	-
		Iterations SEM	-	0.00953277	0.325244	-
		Min Iterations	-	155	7144	-
		Max Iterations	-	256	10529	-
		Duration mean	-	0.448348	21.3664	-
		Duration stdDev	-	0.0476638	1.78143	-
		Duration SEM	-	0.00953277	0.325244	-
		Min Duration	-	0.354125	16.8049	-
		Max Duration	-	0.579428	23.4937	-

Taula 5.4: Resultats per a la funció colville

Funció	Mínim		GA	DE	PSO	ABC
zakharov	0	Fitness mean	0.00234309	0	0	0
		Fitness stdDev	0.00144625	0	0	0
		Fitness SEM	0.000264047	0	0	0
		Min Fitness	0.000440821	0	0	0
		Max Fitness	0.0061645	0	0	0
		Success Ratio	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	409.516	1071.16	29051.9
		Iterations stdDev	-	14.176	47.2182	1152.86
		Iterations SEM	-	0.0107744	0.0336067	0.881364
		Min Iterations	-	374	965	27122
		Max Iterations	-	432	1164	32250
		Duration mean	-	1.55497	4.79841	189.605
		Duration stdDev	-	0.0590137	0.184072	4.82743
		Duration SEM	-	0.0107744	0.0336067	0.881364
		Min Duration	-	1.39925	4.33564	179.372
		Max Duration	-	1.67558	5.1196	200.038

Taula 5.5: Resultats per a la funció Zakharov

Funció	Mínim		GA	DE	PSO	ABC
schwefel2p22	0	Fitness mean	0.0018812	0	0	0
		Fitness stdDev	0.000569504	0	0	0
		Fitness SEM	0.000103977	0	0	0
		Min Fitness	0.000858752	0	0	0
		Max Fitness	0.00314844	0	0	0
		Success Ratio	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	1330.32	2156.9	1464.32
		Iterations stdDev	-	26.5966	47.5417	37.0644
		Iterations SEM	-	0.0264665	0.0591467	0.045011
		Min Iterations	-	1290	2087	1396
		Max Iterations	-	1387	2240	1531
		Duration mean	-	8.28921	20.3926	11.0592
		Duration stdDev	-	0.144963	0.32396	0.246535
		Duration SEM	-	0.0264665	0.0591467	0.045011
		Min Duration	-	8.04419	19.7926	10.5507
		Max Duration	-	8.55065	20.8914	11.5967

Taula 5.6: Resultats per a la funció Schwefel 2p22

Els següent conjunt de funcions afegeix un grau de complexitat a l'hora de realitzar la cerca de l'òptim ja que aquestes són no separables. La funció *Colville*, taula 5.4, ha resultat ser peculiarment difícil d'optimitzar. Tan els GA com els ABC no han aconseguit finalitzar amb èxit cap execució quedant per sobre del límit exigit. Els PSO han completat amb èxit totes les execucions però han requerit un nombre d'iteracions alt, proper a 10.000, comparat amb el que fins ara s'havia observat. Els DE han finalitzat amb èxit la majoria de les execucions, el 83.87%, amb una mitjana d'aproximadament 194 iteracions.

Per a les altres dues funcions, *Zakharov* i *Schwefel 2p22*, els estadístics són els de les taules 5.5 i 5.6 els resultats són similars els que ja s'han comentat per a les funcions *Sphere* i *SumSquares*, on els PSO, DE i ABC finalitzen amb èxit totes les execucions i els GA queden uns quants ordres de magnitud per sobre del mínim. L'única cosa que cal remarcar és l'elevat nombre d'iteracions que ha requerit l'algoritme ABC per finalitzar amb èxit l'optimització de la funció Zakharov, comparat amb les utilitzades en els algoritmes DE i PSO.

Un cop arribat aquest punt, es el torn de analitzar els resultats per a les funcions multimodals. Començant primer per les classificades com a *separables*.

Funció	Mínim		GA	DE	PSO	ABC
Himmelblau	0	Fitness mean	6.6e-09	0	0	0
		Fitness stdDev	2.4e-08	0	0	0
		Fitness SEM	4.4e-09	0	0	0
		Min Fitness	0	0	0	0
		Max Fitness	1.331e-07	0	0	0
		Success Ratio	67.74%	100.00%	100.00%	100.00%
		Iterations mean	1703.29	83.5806	120.645	119.871
		Iterations stdDev	4848.03	20.3339	11.6662	23.5684
		Iterations SEM	0.578177	0.00718236	0.00414188	0.0130571
		Min Iterations	24	59	97	74
		Max Iterations	20602	133	148	161
		Duration mean	0.956115	0.141082	0.154228	0.373287
		Duration stdDev	2.58569	0.0393394	0.022686	0.0715168
		Duration SEM	0.578177	0.00718236	0.00414188	0.0130571
		Min Duration	0.031539	0.0793889	0.108332	0.230992
		Max Duration	10.9127	0.25374	0.188883	0.495292

Taula 5.7: Resultats per a la funció Himmelblau

La funció himmelblau és una de les més senzilles d'aquesta categoria donat el reduït nombre de dimensions que la composen i a la distribució dels seus mínims. Els algoritmes DE, PSO i ABC mostren un 100% d'èxits en la seva execució així com un reduït nombre d'iteracions per a convergir al mínim. Els GA en canvi, han conclòs amb èxit el 67.74% de les execucions quedant amb un mínim mitjà un ordre de magnitud superior al requerit.

Funció	Mínim		GA	DE	PSO	ABC
rastrigin	0	Fitness mean	2.82037e-05	12.1482	98.3883	0
		Fitness stdDev	2.06797e-05	2.85399	17.5643	0
		Fitness SEM	3.7756e-06	0.521064	3.20679	0
		Min Fitness	4.2849e-06	7.95967	62.6823	0
		Max Fitness	0.000104707	18.9042	132.329	0
		Success Ratio	0.00%	0.00%	0.00%	100.00%
		Iterations mean	-	-	-	1123.26
		Iterations stdDev	-	-	-	56.5189
		Iterations SEM	-	-	-	0.0720459
		Min Iterations	-	-	-	1010
		Max Iterations	-	-	-	1216
		Duration mean	-	-	-	11.2374
		Duration stdDev	-	-	-	0.394611
		Duration SEM	-	-	-	0.0720459
		Min Duration	-	-	-	10.4161
		Max Duration	-	-	-	11.8147

Taula 5.8: Resultats per a la funció Rastrigin

Els resultats de l'optimització de la funció *rastrigin*, taula 5.8, mostren com l'únic algoritme que ha finalitzat amb èxit les execucions és l'ABC. L'algoritme GA no mostra suficient velocitat de convergència i queda relativament allunyat del mínim desitjat. Tan l'algoritme DE com el PSO mostren un resultats molt allunyats del mínim, sobre tot l'algoritme PSO, fet que fa suposar que hi ha algun problema de convergència cap a mínims locals.

Resultats similars són els obtinguts per a la funció *schwefel*, taula 5.9. A més en aquest cas, les diferències entre el valor desitjat i l'obtingut amb els algoritmes DE i PSO són majors que en el cas de la funció *rastrigin*, ja que la funció *schwefel*, com

s'ha exposat a l'apartat 4.1.3, es caracteritza per tenir un mínim global geomètricament distant, en l'espai de paràmetres, dels millors mínims locals, facilitant d'aquesta manera la convergència a un mínim no desitjat.

Funció	Mínim		GA	DE	PSO	ABC
schwefel	-12569.5	Fitness mean	-12569.5	-10605.8	-8850.04	-12569.5
		Fitness stdDev	0.000119035	556.113	426.632	0
		Fitness SEM	2.17328e-05	101.532	77.8921	0
		Min Fitness	-12569.5	-11500.5	-9899.86	-12569.5
		Max Fitness	-12569.5	-9235.01	-8261.56	-12569.5
		Success Ratio	0.00%	0.00%	0.00%	100.00%
		Iterations mean	-	-	-	4585.77
		Iterations stdDev	-	-	-	475.337
		Iterations SEM	-	-	-	0.442647
		Min Iterations	-	-	-	3621
		Max Iterations	-	-	-	5573
		Duration mean	-	-	-	35.3591
		Duration stdDev	-	-	-	2.42448
		Duration SEM	-	-	-	0.442647
		Min Duration	-	-	-	29.5958
		Max Duration	-	-	-	39.4878

Taula 5.9: Resultats per a la funció Schwefel

Funció	Mínim		GA	DE	PSO	ABC
michalewicz5	-4.68766	Fitness mean	-4.68766	-4.68147	-4.68631	-4.68766
		Fitness stdDev	5.8e-08	0.0272731	0.00737885	0
		Fitness SEM	1.06e-08	0.00497937	0.00134719	0
		Min Fitness	-4.68766	-4.68766	-4.68766	-4.68766
		Max Fitness	-4.68766	-4.53766	-4.6459	-4.68766
		Success Ratio	35.48%	93.55%	96.77%	100.00%
		Iterations mean	22231.4	175.483	1134.2	1524.45
		Iterations stdDev	13791.3	14.2239	2610.55	309.891
		Iterations SEM	6.09553	0.00693078	0.631907	0.18743
		Min Iterations	3585	153	262	1048
		Max Iterations	46381	208	14824	2511
		Duration mean	32.6839	0.526027	2.13602	7.9267
		Duration stdDev	19.2758	0.0366743	3.40292	1.0266
		Duration SEM	6.09553	0.00693078	0.631907	0.18743
		Min Duration	6.20305	0.451409	0.743731	5.82652
		Max Duration	60.2171	0.602215	19.8084	10.6969

Taula 5.10: Resultats per a la funció Michalewicz 5

Els resultats de l'optimització de la funció *Michalewicz 5*, taula 5.10, mostren com tots els algoritmes han finalitzat en major o menor mesura alguna de les seves execucions amb èxit. El millor ha estat l'algoritme ABC ja que ha finalitzat totes les execucions amb èxit. Tan l'algoritme DE com el PSO en la majoria de casos han finalitzat amb èxit. Només en 2 casos l'algoritme DE no han trobat el mínim i en un cas no ho ha fet el PSO. Tot hi que l'algoritme GA té un nombre d'èxits molt menor als dels dos algoritmes anteriors, el valor mitjà obtingut és millor, degut a la penalització que comporta convergir en un mínim local tal com ho han fet l'algoritme DE i el PSO en un nombre limitat d'ocasions.

L'optimització de la funció *shaffer* deixa uns resultats similars, taula 5.11, a la funció acabada de comentar. Tan l'algoritme ABC i el PSO convergeixen a l'òptim en totes les seves execucions, emprant menys iteracions en el cas del primer. L'algoritme GA torna donar mostra de la seva lentitud a l'hora de convergir cap a l'òptim, finalitzant només en

una ocasió l'execució en èxit. L'algoritme DE finalitza amb èxit el 58.06%, mostrant una ràpida convergència en el cas d'aquests èxits, unes 200 iteracions de mitjana.

Funció	Mínim		GA	DE	PSO	ABC
schaffer	0	Fitness mean	0.00908916	0.00378027	0	0
		Fitness stdDev	0.00238659	0.00471832	0	0
		Fitness SEM	0.000435729	0.000861444	0	0
		Min Fitness	0	0	0	0
		Max Fitness	0.00971591	0.00971591	0	0
		Success Ratio	3.23%	58.06%	100.00%	100.00%
		Iterations mean	15069	200.556	1185	430.194
		Iterations stdDev	0	35.0115	1891.82	158.297
		Iterations SEM	0	0.0151254	0.249774	0.0777211
		Min Iterations	15069	151	138	183
		Max Iterations	15069	280	8453	728
		Duration mean	21.3282	0.433902	1.12724	1.49176
		Duration stdDev	0	0.0623635	1.36807	0.425696
		Duration SEM	0	0.0151254	0.249774	0.0777211
		Min Duration	21.3282	0.332504	0.247715	0.669367
		Max Duration	21.3282	0.590151	6.37583	2.17058

Taula 5.11: Resultats per a la funció schaffer

Funció	Mínim		GA	DE	PSO	ABC
shubert	-186.731	Fitness mean	-186.731	-186.731	-186.731	-186.315
		Fitness stdDev	1.27526e-05	0	0	0.37353
		Fitness SEM	2.3283e-06	0	0	0.068197
		Min Fitness	-186.731	-186.731	-186.731	-186.712
		Max Fitness	-186.731	-186.731	-186.731	-185.201
		Success Ratio	61.29%	100.00%	100.00%	0.00%
		Iterations mean	3837.21	196.29	152.839	-
		Iterations stdDev	10449.9	32.9018	15.3562	-
		Iterations SEM	1.40395	0.0129232	0.00546031	-
		Min Iterations	38	121	122	-
		Max Iterations	43939	273	183	-
		Duration mean	2.3968	0.477718	0.288316	-
		Duration stdDev	5.95646	0.0707831	0.0299073	-
		Duration SEM	1.40395	0.0129232	0.00546031	-
		Min Duration	0.055645	0.278245	0.234286	-
		Max Duration	24.9251	0.601408	0.337643	-

Taula 5.12: Resultats per a la funció Shubert

La funció *shubert*, multimodal i no separable, mostra, taula 5.12 una convergència a l'òptim en totes les execucions dels algoritmes DE i PSO amb un nombre d'iteracions necessàries baix. En el cas de l'algoritme aquest nombre GA és més elevat, unes 3837. L'algoritme ABC no conclou amb èxit cap de les execucions, donant indicis de una convergència en un òptim local.

Els pròxims resultats a comentar, els de l'optimització de la funció *griewank* (taula 5.13), mostren com l'algoritme ABC és l'únic que conclou el procés de cerca de manera favorable. Els algoritmes DE i PSO troben l'òptim en un nombre elevat d'execucions mostrant els segons un major temps de convergència. Els GA no finalitzen cap execució amb èxit.

La cerca de l'òptim en el cas de la funció *ackley* deixa uns resultats molt similars, taula 5.14, els comentats per a la funció *sphere*. Els algoritmes DE, PSO i ABC finalitzen

el 100% dels casos amb èxit. En cap cas l'algoritme GA a trobat l'òptim amb la precisió desitjada.

Funció	Mínim		GA	DE	PSO	ABC
griewank	0	Fitness mean	0.0295564	0.00103325	0.00119243	0
		Fitness stdDev	0.0281282	0.00317715	0.0031835	0
		Fitness SEM	0.00513549	0.000580065	0.000581226	0
		Min Fitness	1.77882e-05	0	0	0
		Max Fitness	0.146416	0.0123161	0.0123161	0
		Success Ratio	0.00%	90.32%	87.10%	100.00%
		Iterations mean	-	860.321	4021.7	1155.1
		Iterations stdDev	-	25.8347	8646.32	128.403
		Iterations SEM	-	0.0263702	8.78411	0.148691
		Min Iterations	-	825	1413	993
		Max Iterations	-	940	40150	1526
		Duration mean	-	6.22317	28.8359	10.8156
		Duration stdDev	-	0.137024	44.7903	0.814412
		Duration SEM	-	0.0263702	8.78411	0.148691
		Min Duration	-	5.97131	14.8433	9.66174
		Max Duration	-	6.52498	214.605	12.932

Taula 5.13: Resultats per a la funció Griewank

Funció	Mínim		GA	DE	PSO	ABC
ackley	0	Fitness mean	0.00174528	0	0	0
		Fitness stdDev	0.000720476	0	0	0
		Fitness SEM	0.00013154	0	0	0
		Min Fitness	0.000559891	0	0	0
		Max Fitness	0.00338645	0	0	0
		Success Ratio	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	1328.29	2790.9	1590.16
		Iterations stdDev	-	28.3631	693.452	35.0779
		Iterations SEM	-	0.0321281	0.700774	0.0569696
		Min Iterations	-	1278	2378	1513
		Max Iterations	-	1394	6277	1656
		Duration mean	-	9.84316	28.2649	15.4484
		Duration stdDev	-	0.175973	3.8383	0.312035
		Duration SEM	-	0.0321281	0.700774	0.0569696
		Min Duration	-	9.51041	25.5275	14.7142
		Max Duration	-	10.1375	47.0858	16.1023

Taula 5.14: Resultats per a la funció Ackley

Funció	Mínim		GA	DE	PSO	ABC
shekel10	-10.5364	Fitness mean	-6.99304	-10.2891	-10.5364	-4.49987
		Fitness stdDev	3.45469	1.35434	0	1.96929
		Fitness SEM	0.630736	0.247267	0	0.359542
		Min Fitness	-10.5364	-10.5364	-10.5364	-9.11186
		Max Fitness	-2.80663	-2.87114	-10.5364	-1.99334
		Success Ratio	19.35%	96.77%	100.00%	0.00%
		Iterations mean	24584.5	124.167	229.226	-
		Iterations stdDev	11253.7	9.85929	21.3416	-
		Iterations SEM	6.76918	0.00749503	0.0163209	-
		Min Iterations	7414	109	190	-
		Max Iterations	38774	148	277	-
		Duration mean	33.8067	0.525284	0.914309	-
		Duration stdDev	15.1364	0.0403619	0.0893933	-
		Duration SEM	6.76918	0.00749503	0.0163209	-
		Min Duration	11.4433	0.445307	0.742144	-
		Max Duration	56.9281	0.605297	1.08556	-

Taula 5.15: Resultats per a la funció Shekel10

Els últims estadístics a comentar són els de l'optimització de la funció *shekel10*, taula 5.15. L'únic algoritme que en totes les execucions troba l'òptim és el PSO. En un nombre elevat de vegades, el 96.77%, l'algoritme DE també compleix aquest objectiu. L'algoritme GA en 6 ocasions també troba el mínim, però aquests a costa d'un nombre elevat d'iteracions, una mitjana de 24584.5. L'algoritme ABC no culmina el procés de cerca en cap de les seves execucions, quedant molt per sobre de l'objectiu marcat.

5.1.1 Anàlisi dels estadístics

	GA	DE	PSO	ABC
sphere	—	•	•	•
step	•	•	•	•
sumSquares	—	•	•	•
colville	—	∘ (83.87%)	•	—
zakharov	—	•	•	•
schwefel 2p22	—	•	•	•
himmelblau	∘ (67.74%)	•	•	•
rastrigin	—	—	—	•
schwefel	—	—	—	•
michalewicz5	∘ (35.48%)	∘ (93.55%)	∘ (96.77%)	•
schaffer	∘ (3.23%)	∘ (58.06%)	•	•
shubert	∘ (61.29%)	•	•	–
griewank	–	∘ (90.32%)	∘ (87.10%)	•
ackley	—	•	•	•
shekel10	∘ (19.35%)	∘ (96.77%)	•	–

Taula 5.16: Resum dels resultats

La taula 5.16 resumeix els resultats de l'optimització de cada una de les funcions realitzada pels diversos algoritmes. En aquesta taula, el símbol • representa 100% d'execucions finalitzades amb èxit, el símbol ∘ representa que un algoritme determinat ha trobat l'òptim en algunes execucions i, per finalitzar, el símbol — representa la no consecució de l'objectiu de cerca en cap ocasió.

A continuació es donaran unes quantes observacions sobre el comportament individual[1] de cada un dels algoritmes de computació evolutiva utilitzats en aquest PFC.

Observacions sobre el comportament de l'algoritme GA

L'algoritme GA a demostrat tenir una lenta convergència cap a l'òptim de la funció que està optimitzant. A la gran majoria d'execucions no ha estat capaç trobar el mínim amb la precisió estipulada per a l'experiment i en les que ho ha fet, ha requerit un nombre d'iteracions elevat.

En quan el problema de la convergència a mínims locals, l'algoritme s'ha mostrat prou robust en la majoria d'execucions. Tan sols es detecten indicis d'aquest tipus de convergència no desitjada en algunes de les execucions utilitzant la funció *shekel10*. Aquesta funció és caracteritza per donar molt poca informació al procés de cerca ja que els mínims,

[1] A la següent secció, 5.2, es mostrara l'anàlisi comparatiu entre els diversos algoritmes

tan locals com global, estan repartits en zones molt restringides, veure figura 4.1l.

Aquests dos fets apunten a que el procés de cerca de l'algoritme GA no compensa els modes d'exploració i explotació, donant prioritat al primer. Això es pot deure al tipus d'operador de mutació utilitzat, mutació uniforme d'una dimensió dins el rang de cerca estipulat, ja que és aquest el que controla en part l'introducció de diversitat a la mostra. Utilitzar una mutació basada en alguna distribució podria corregir aquest problema, però aquesta tècnica requereix una configuració especifica depenent del domini de la cerca utilitzat i un major temps computacional. Una opció més complexa és l'utilització de l'operador de mutació descrit per Deep i Thakur [18].

Observacions sobre el comportament de l'algoritme DE

Aquest algoritme és el que ha mostrat més problemes a l'hora d'evitar la convergència a un òptim local. Aquest fet és pot comprovar observant el nombre màxim d'iteracions que ha necessitat a l'hora d'optimitzar les funcions per aquells casos on només ha conclòs amb èxit algunes de les seves execucions, marcades amb el símbol $\circ$ a la taula 5.16. Es pot veure com l'algoritme o convergeix a l'òptim en relativament poques iteracions, 1394 iteracions l'execució que més n'ha precisat (*acley*), o no convergeix. A més, el nombre d'iteracions necessàries per optimitzar una funció varia poc entre les diverses execucions, com es pot comprovar amb els valors de la desviació estàndard recollits a les taules de resultats.

Un mecanisme per intentar evitar aquesta convergència prematura és introduir n noves solucions candidates, amb les corresponents dimensions seleccionades de manera uniformement aleatòria dins el rang de cerca estipulat, cada x iteracions eliminant prèviament les solucions menys aptes presents a la població. D'aquesta manera, estimularíem el mode de exploració a costa d'un augment en el temps de convergència. El principal problema d'aquest mètode és trobar uns valors n i x adients.

Observacions sobre el comportament de l'algoritme PSO

El primer que cal destacar d'aquest algoritme és el bon comportament obtingut a l'hora d'optimitzar funcions unimodals, ha trobat el mínim a les sis funcions d'aquest tipus presents a l'experiment.

Més resistència han oposat les funcions *rastrigin* i *schwefel*, ambdues multimodals i amb un elevat nombre de dimensions, trenta. L'algoritme no ha mostrat cap fi satisfactori a l'hora d'optimitzar-les, quedant molt per sobre del valor mínim requerit. Aquest fet pareix indicar que hi ha una convergència prematura en algun mínim local de la que l'algoritme no s'ha pogut recuperar.

A diferència de l'observat per l'algoritme DE, el nombre d'iteracions mitjanes per a trobar l'òptim desitjat no té perquè ser baix, e.g. s'han necessitat, en el pitjor dels casos, 40150 iteracions per completar satisfactòriament l'optimització de la funció *griewank* (veure taula 5.13). A més, la variabilitat del nombre d'iteracions necessàries per convergir

és considerablement més elevada que per l'algoritme anteriorment comentat.

Per intentar dotar d'una major resistència a la convergència prematura, es pot canviar la topologia utilitzada, la manera com les partícules interactuen amb les altres, utilitzant patrons més complexos. Per exemple, seguint les indicacions de Kennedy [36] es pot optar per utilitzar la topologia Von Neumann amb la qual s'han obtingut bons resultats en proves experimentals. Una altre solució seria controlar la distància a la que les partícules poden apropar-se, repel·lent-se si es sobrepassa un determinat llindar. S'ha de tenir en compte que aquestes propostes tenen un cost computacional major que l'implementació utilitzada.

Observacions sobre el comportament de l'algoritme ABC

A diferència dels anteriors algoritmes, l'ABC no mostra resultats positius parcials: O té un 100% encerts a l'hora d'optimitzar una funció o no l'optimitza en cap cas.

Observant quines han estat les funcions que no ha aconseguit optimitzar (*colville*, *shubert* i *shekel10*), es posa de manifest com aquestes són funcions amb un baix nombre de dimensions. Aquest fet pot indicar que el procediment utilitzat per calcular el desplaçament pot ser no idoni per aquest tipus de funcions ja que no guia amb suficient eficàcia el procés d'explotació.

5.2 Comparativa dels resultats

Funció	t-value	p-value	Rànquing	Significació	Millor algoritme
step	1	0.321327	1	0.05	-
himmelblau	-1.50302	0.138078	2	0.025	-
michalewicz5	-2.28651	0.0257689	3	0.0166667	-
shekel10	3.43406	0.00108398	4	0.0125	GA
colville	-5.29682	1.767e-06	5	0.01	ABC
griewank	-5.75532	3.136e-07	6	0.00833333	ABC
shubert	6.09228	8.61e-08	7	0.00714286	GA
sphere	-7.092	1.7e-09	8	0.00625	ABC
rastrigin	-7.47005	4e-10	9	0.00555556	ABC
schwefel	-7.68664	2e-10	10	0.005	ABC
zakharov	-8.87376	0	11	0.00454545	ABC
sumSquares	-11.0069	0	12	0.00416667	ABC
ackley	-13.268	0	13	0.00384615	ABC
schwefel2p22	-18.0926	0	14	0.00357143	ABC
schaffer	-20.8596	0	15	0.00333333	ABC

Taula 5.17: Comparativa ABC vs GA

La taula 5.17, mosta la comparativa de l'eficàcia a l'hora de cercar el mínim de les diverses funcions per l'algoritme GA i l'algoritme ABC. Es pot veure com en aquest cas, l'algoritme ABC és significativament millor que l'algoritme GA a l'hora d'optimitar la majoria de funcions. Tan sols és superat pel GA en els casos de les funcions *shekel10*

i *shubert*. No hi ha hagut diferència significativa en l'eficiència en les funcions *step*, *himmelblau* i *michalewicz5*.

Funció	t-value	p-value	Rànquing	Significació	Millor algoritme
sphere	1	0.321327	1	0.05	-
step	1	0.321327	2	0.025	-
sumSquares	1	0.321327	3	0.0166667	-
zakharov	1	0.321327	4	0.0125	-
schwefel2p22	1	0.321327	5	0.01	-
himmelblau	1	0.321327	6	0.00833333	-
ackley	1	0.321327	7	0.00714286	-
michalewicz5	-1.24232	0.218953	8	0.00625	-
colville	-1.55155	0.126028	9	0.00555556	-
griewank	-1.78126	0.0799316	10	0.005	-
schaffer	-4.3883	4.69902e-05	11	0.00454545	ABC
shubert	6.09235	8.61e-08	12	0.00416667	DE
shekel10	13.2672	0	13	0.00384615	DE
schwefel	-19.3406	0	14	0.00357143	ABC
rastrigin	-23.3142	0	15	0.00333333	ABC

Taula 5.18: Comparativa ABC vs DE

L'eficàcia de l'algoritme DE i ABC és similar per a la majoria de funcions, taulas 5.18. En aquest cas només s'aprecia una diferència significativa a l'hora d'optimitzar 5 funcions. A favor d'ABC: *shaffer*, *schwefel* i *rastrigin*. I a favor de DE *shubert* i *shekel10*.

Similars resultats són els mostrats a la comparativa de ABC i PSO, taula 5.19. En aquest cas també hi ha només diferències significatives en l'optimització de 5 funcions. A favor d'ABC: *rastrigin* i *schwefel*. A favor d'PSO: *colville*, *shekel10* i *griewank*.

Funció	t-value	p-value	Rànquing	Significació	Millor algoritme
sphere	1	0.321327	1	0.05	-
step	1	0.321327	2	0.025	-
sumSquares	1	0.321327	3	0.0166667	-
zakharov	1	0.321327	4	0.0125	-
schwefel2p22	1	0.321327	5	0.01	-
himmelblau	1	0.321327	6	0.00833333	-
schaffer	1	0.321327	7	0.00714286	-
ackley	1	0.321327	8	0.00625	-
michalewicz5	-1	0.321327	9	0.00555556	-
griewank	-2.05158	0.0445825	10	0.005	-
shubert	6.09235	8.61e-08	11	0.00454545	PSO
colville	11.8672	0	12	0.00416667	PSO
shekel10	16.7895	0	13	0.00384615	PSO
rastrigin	-30.6812	0	14	0.00357143	ABC
schwefel	-47.7513	0	15	0.00333333	ABC

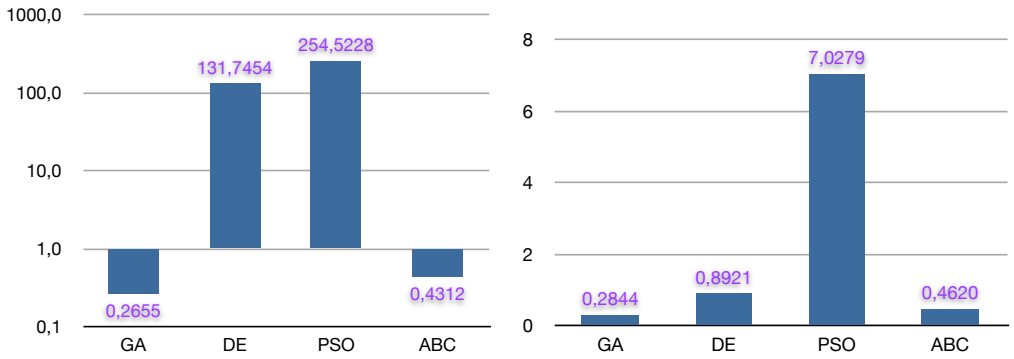
Taula 5.19: Comparativa ABC vs PSO

La taula 5.20 mostra un resum de les diverses comparatives que s'acaben d'exposar.

ABC *vs*	GA	DE	PSO
sphere	ABC	-	-
step	-	-	-
sumSquares	ABC	-	-
colville	ABC	-	PSO
zakharov	ABC	-	-
schwefel 2p22	ABC	-	-
himmelblau	-	-	-
rastrigin	ABC	ABC	ABC
schwefel	ABC	ABC	ABC
michalewicz5	-	-	-
schaffer	ABC	ABC	-
shubert	GA	DE	PSO
griewank	ABC	-	-
ackley	ABC	-	-
shekel10	GA	DE	PSO

Taula 5.20: Resum de comparatives

Una altre mesura que podem comparar és la mitjana d'errors absoluts, tal com fa Karaboga [32]. Per fer aquest càlcul, es comprova quina és la diferència que hi ha entre el mínim real i el mínim trobat per l'algoritme per cada una de les funcions. Després, es calcula la mitjana d'aquests valors per a cada algoritme. La figura 5.1a mostra el resultat d'aquest procés. Es pot veure que l'algoritme que menys error mitjà té és l'algoritme GA, seguit pel ABC. En canvi els algoritmes DE i PSO mostren errors mitjans més elevats.



(a) Totes les funcions **(b)** Sense incloure la funció schwefel

Figura 5.1: Mitjana d'errors absoluts

S'ha de remarcar que els resultats depenen molt del conjunt de funcions seleccionat i de a quina d'aquestes funcions un determinat algoritme ha convergit a un òptim local durant la seva optimització. Això és així ja que hi ha funcions que, donades les seves propietats, penalitzen més que les altres la convergència a mínims locals. La funció schwefel és una d'aquestes. Si es realitza el mateix càlcul de l'error mitja sense tenir en compte aquesta funció, figura 5.1b, es pot veure com les diferències entre els algoritmes es redueixen de manera significativa ja que tan l'algoritme DE com el PSO havien convergit en un mínim local de la funció schwefel.

Capítol 6

Conclusions

Un dels objectius d'aquest PFC era exposar els fonaments de la computació evolutiva. Al capítol 2 s'han introduït i descrit les principals branques d'aquesta disciplina, els EA i els algoritmes basats en SI. A més, s'ha donat una visió de l'evolució d'aquest cap el llarg de l'historia exposant quins han estats els principals autors que hi han intervengut. Per finalitzar aquest capítol, s'han descrit els principals camps d'utilització d'aquest tipus de tècniques.

Acte seguit, capítol 3, s'han descrit els algoritmes utilitzats en aquest experiment (GA, DE, PSO i ABC), exposant per a cada un d'ells el funcionament teòric i les decisions que respecte el seu funcionament s'han pres. Així mateix, es detallen quin són els valors seleccionats pels diversos paràmetres de configuració de cada algoritme: rati de mutació, influència dels veïns, constricció simple...

Al capítol 4 s'ha introduït el conjunt de funcions a optimitzar. S'ha detallat la formulació, el nombre de dimensions, el rang de cerca i la categoria en termes de modalitat i separabilitat de cada una de les funcions. En aquest mateix capítol s'ha exposat la metodologia utilitzada per a la realització de l'experiment i la comparativa dels resultats dels resultats obtinguts.

Els resultats de l'experiment estan desglossats al capítol 5. Primer s'han detallat els estadístics del procés de optimització per cada funció i algoritme i després s'ha realitzat una comparativa entre l'eficàcia de l'algoritme ABC comparat amb els altres algoritmes. De l'anàlisi d'aquests resultat es desprenen diverses conclusions.

L'algoritme GA no mostra indicis de convergència a òptims locals però si mostra un lenta convergència cap el mínim global degut al desequilibri entre el mode d'exploració i el mode d'explotació.

L'algoritme DE es mostra propens a la convergència prematura a mínims locals sense que pugui restablir el procés de cerca un cop ha convergit. El fet que l'algoritme o trobi el resultat òptim en poques iteracions o no el trobi demostra l'anteriorment esmentat.

El PSO mostra un bon comportament per a funcions unimodals, però quan intenta

optimitzar funcions amb mínims locals i alt nombre de dimensions es propens a convergir prematurament. Contràriament a l'exposat per l'algoritme DE, el nombre d'iteracions per trobar l'òptim no te perquè ser baix, fet que pot indicar cert grau de recuperació davant la convergència a un mínim no global.

L'algoritme ABC, s'ha mostrat solid a l'hora de optimitzar la majoria de funcions però ha tingut problemes a l'hora d'optimitzar funcions amb baix nombre de dimensions, no trobat per aquestes el mínim desitjat.

La comparativa entre l'algoritme ABC i la resta mostra resultats que confirmen el que s'acaba de dir. La lenta convergència de l'algoritme GA no l'ha permès assolir el mínim teòric amb la suficient precisió, fet que com mostra la taula 5.17, la penalitzat a favor de l'ABC. Les taules 5.18 i 5.19 mostren les comparatives de l'eficàcia entre l'ABC i els algoritmes DE i PSO on també es posa de manifest que el primer és més eficaç amb algunes de les funcions més complexes i falla a l'hora d'optimitzar diverses funcions amb un nombre baix d'iteracions.

6.1 Possibles millores

Diverses han estat les solucions que, per intentar resoldre aquests problemes dels diferents algoritmes, s'han proposat a l'apartat 5.1.1:

- Canviar l'operador de mutació de l'algoritme GA per tal d'afavorir una més ràpida convergència a l'òptim.
- En el cas dels DE, introduir noves solucions candidates, generades aleatòriament dins el rang de cerca, quan no es detecti una millora de la solució després de n iteracions.
- Canviar la topologia de les relacions socials de les partícules a l'algoritme PSO per una topologia Von Neumann. També s'ha proposat implementar un mecanisme de repulsió si 2 partícules s'atraquen molt dins l'espai de solucions.
- A l'ABC, revisar el generador de desplaçament per tal d'intentar minimitzar els efectes negatius que aquest té a l'hora d'optimitzar funcions amb poc nombre de dimensions.

A part d'aquestes millores individuals per cada un dels algoritmes es proposa la utilització d'un sistema *híbrid* per tal de generar el conjunt inicial de solucions candidates, la població. Aquesta seria generada per un algoritme de cerca de mínims locals com el descrit per Eiben [20, pag. 175] o De Castro [17], els quals, després d'unes poques iteracions[1] retornen el conjunt trobat perquè sigui l'algoritme seleccionat qui finalitzi el procés de cerca.

[1]Molts d'aquests algoritmes calculen en algun moment la distancia entre les solucions que han trobat, calcul que pot arribar a tenir un cost computacional elevat si el nombre de dimensions és elevat. Per tant resulten poc eficients si es precisa la cerca del mínim global amb molta precisió.

Apèndix A

Detalls de l'implementació del prototipus

A.1 Programari utilitzat

La implementació de l'experiment s'ha realitzat utilitzant el llenguatge de programació Python[1], concretament la versió 2.7[2]. El programari requerit per a la correcte execució de l'experiment és:

- **Debian 6.0**[3]: Sistema operatiu utilitzat.
- **Intèrpret de Python v.2.7**
- **SciPy v.1.6.1**[4]: Paquet que conté nombroses funcions per al càlcul numèric. S'ha utilitzat per a realitzar el càlcul del SEM de les execucions així com per a realitzar el *t-test*.
- **NumPy v.1.6.1**[5]: Un subconjunt d'utilitats del paquet SciPy utilitzat per calcular les mitjanes i desviacions estàndard dels resultats.
- **TerminalProgressBar**[6]: Mostra una barra interactiva amb informació sobre el percentatge d'experiment executat. Donada la seva implementació requereix una versió de Python inferior a la 3.0.
- **MathPlotLib v.1.1.0**[7]: Aquesta llibreria no és necessària per a la correcte execució de l'experiment, de fet, en el codi proporcionat, no hi ha cap crida a aquesta llibreria. No obstant, ha estat utilitzada per crear les representacions de la figura 4.1.

[1] `http://www.python.org/`

[2] L'execució de l'experiment amb una versió del interpret de Python major a la 3.0, pot donar lloc a errors degut a les dependències d'alguns paquets auxiliars.

[3] `http://www.debian.org`

[4] `http://www.scipy.org/`

[5] `http://numpy.scipy.org/`

[6] `http://nadiana.com/animated-terminal-progress-bar-in-python`

[7] `http://matplotlib.sourceforge.net/`

A.2 Maquinaria utilitzada

Donat el nombre total d'execucions a realitzar per a cada un dels algoritmes i el nombre màxim d'iteracions que cada una d'aquestes pot generar, l'execució en un temps acceptable requereix l'ús de maquinaria d'alt rendiment. Afortunadament, Amazon ofereix un servei de computació baix demanda que permet l'accés a aquest tipus de recursos[8]. Dels diferents tipus de configuracions que es poden seleccionar[9] s'ha elegit, donat el seu poder de còmput, la denominada *Cluster Compute Eight Extra Large Instance* amb la següent configuració:

- 60.5 GB de memòria RAM
- 88 EC2 Compute Units[10] (2 x Intel Xeon ES-2670, arquitectura *Sandy Bridge*)
- 3370 GB d'emmagatzemament
- Plataforma 64-bits
- 10 Gigabit Ethernet

A.3 Estructura del prototipus

L'estructura de l'experiment és, gràcies a python, totalment modular. En aquest es pot dividir en els següents mòduls:

- **Experiment**[11]: Aquest mòdul és l'encarregat d'iniciar i controlar les execucions així com el seu anàlisi: nombre d'execucions, paràmetres de les funcions, mostra de resultats... Al següent annex, A.4, és mostraran més detalls.
- **Funcions**[12]: Modul que detallà les característiques de les funcions d'avaluació i la seva implementació, així com altres mètodes per optimitzar el seu càlcul.
- **Resultats**[13]: Conté les representacions dels resultats dels estadístics de les diferents execucions així com la de les diverses comparatives entre algoritmes.
- **Algoritmes**[14]: Objectes que representen l'execució d'un algoritmes computació evolutiva amb les seves característiques.

Abans d'entrar en el detalls de les diferents implementacions, es presenta el diagrama UML de les principals classes que interactuen a l'experiment, figura A.1. En aquest podem veure una referència als atributs i mètodes que els composen.

[8] `http://aws.amazon.com/ec2/`

[9] `http://aws.amazon.com/ec2/instance-types/`

[10] Mesura utilitzada per Amazon per comparar el rendiment dels diversos tipus d'instàncies. Segons Amazon, un EC2 equival a la capacitat de CPU d'un processador 1.0-1.2 GHz Opteron o Xeon del 2007

[11] Contingut dins l'arxiu experiment.py

[12] Arxiu functions.py

[13] Arxiu results.py

[14] Arxius continguts dins la carpeta algorithms

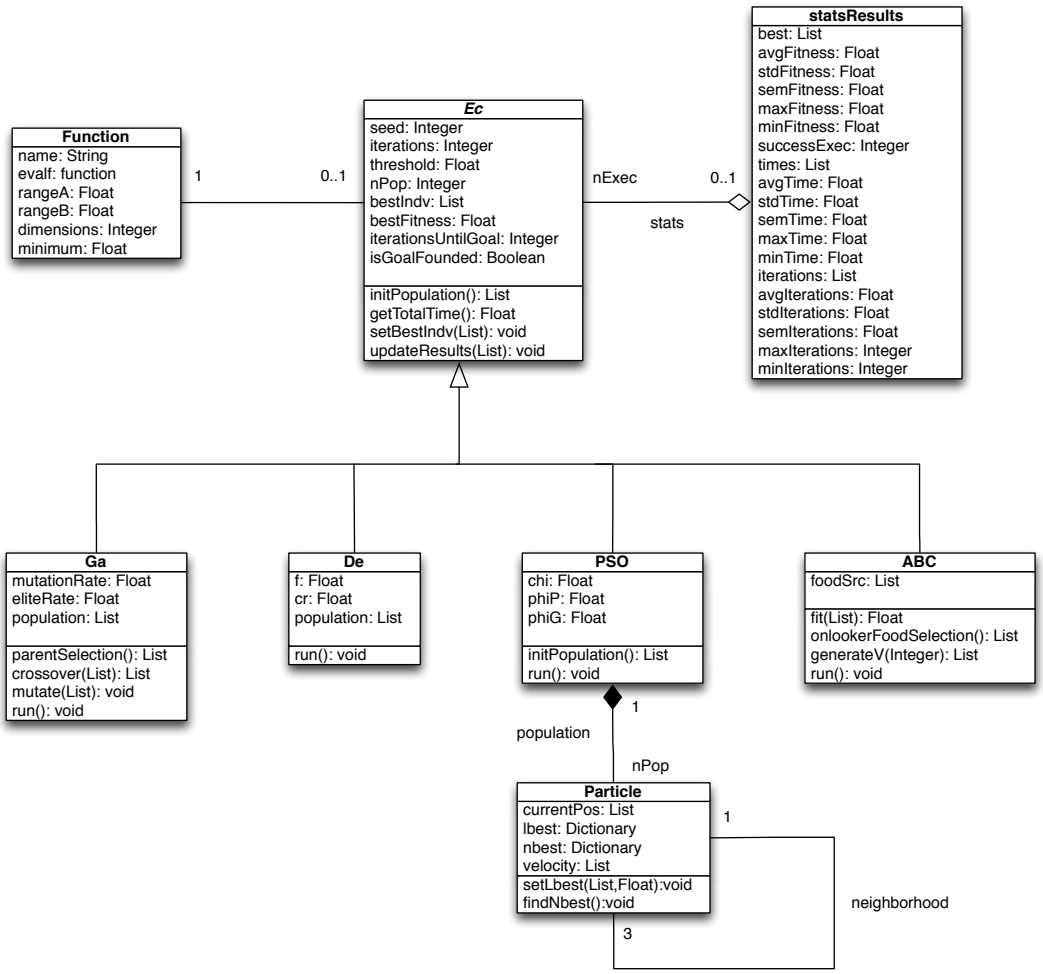


Figura A.1: UML de classes que interactuen a l'experiment

A.4 Implementació de l'experiment

Ja s'ha esmentat que hi ha un mòdul encarregat de gestionar l'execució de l'experiment i que aquest està implementat dins l'arxiu adjunt *experiment.py*. A continuació s'explicarà el seu funcionament desglossant-lo en les diferents funcions que el composen.

La funció principal que controla l'experiment és **main**(*argv*). Comença comprovant els paràmetres d'entrada que formaran la configuració de l'experiment, si no s'han donat, els hi assignarà el valor per defecte. Aquests paràmetres són:

- Nombre d'execucions per algoritme i funció.
- Llavor del generador pseudoaleatori.
- Nombre d'iteracions màximes per execució.
- Detallar el fitxer de sortida amb els resultats de les execucions.

- Detallar el fitxer d'entrada amb els resultats d'una execució anterior. Si aquest paràmetre es present a la crida el programa no executa les execucions, simplement mostra els resultats per pantalla.
- *Output* amb les taules amb els resultats en format LaTeX.

Si és crida el programa amb el paràmetre h[15] es mostrarà per sortida estàndard un resum de les opcions disponibles, figura A.2

```
Aquestes són les opcions disponibles per iniciar l'execució de l'experiment:

    -v\--verbose           Mostra els estadistics de les diferents execucions per pantalla
    -i\--iterations=[0-9]* Nombre màxims d'iteracions per execució. Per defecte 50.000
    -e\--numExec=[0-9]*    Nombre d'execucions per funció i algoritme. Per defecte 31
    -s\--seed=[0-9]*       Llavor del generador pseudoaleatori. Per defecte 42
    -i\--input={filename}  Permet seleccionar un fitxer d'entrada amb dades sobre una execució anterior.
    -o\--output={filename} Crea un fitxer amb les dades dels objectes de l'execució
    -l\--latex             Crea fitxers amb els resultats en format LaTeX

         for help use --help
```

Figura A.2: Opcions disponibles per a l'execució de l'experiment

Un cop comprovats aquests paràmetres, inicia els objectes que representen les diferents funcions d'optimització amb la seva respectiva configuració, codi A.1.

```
functions = [function('sphere', sphere, 30, -100, 100, 0),
             function('step',step,30,-100,100,0),
             function('sumSquares',sumSquares,30,-10,10,0),
             function('colville',colville,4,-10,10,0),
             function('zakharov',zakharov,10,-5,10,0),
             function('schwefel2p22',schwefel2p22,30,-10,10,0),
             function('himmelblau',himmelblau,2,-10,10,0),
             function('rastrigin',rastrigin,30,-5.12,5.12,0),
             function('schwefel',schwefel,30,-500,500,-12569.4866181730),
             function('michalewicz5',michalewicz5,5,0,math.pi,-4.6876581790),
             function('schaffer',schaffer,2,-100,100,0),
             function('shubert',shubert,2,-10,10, -186.7309088310),
             function('griewank',griewank,30,-600,600,0),
             function('ackley',ackley,30,-32,32,0),
             function('shekel10',shekel10,4,0,10,-10.5364098167)]
```

Listing A.1: Ini. de les funcions

Un cop s'han inicialitzat les funcions, ja es pot procedir a iniciar les execucions. Hi ha n[16] execucions per algoritme i funció. Donat que aquestes execucions són independents, s'ha optat per una implementació multiprocés per tal d'optimitzar el temps d'execució de l'experiment mitjançant l'ús dels diferents nuclis de procés presents als processadors actuals.

[15] python experiment.py -h
[16] Per defecte 31

Python utilitza la llibreria multiprocessing[17] per implementar el multiprocés donant per a tal efecte diverses utilitats. La funció **exe(***algo, function, s, ite, nExec***)** és l'encarregada de gestionar aquest procés:

```
def exe(algo, function, s, ite, nExec):
    manager = Manager()
    mutex = manager.Lock()
    semaphore = manager.BoundedSemaphore(cpu_count())
    results = manager.list()

    processes = [Process(target=exeProcess, args=(algo, function, s+i, ite, mutex,\\
                                semaphore, results)) for i in range(nExec)]

    for process in processes:
        semaphore.acquire()
        process.start()

    while len(results) < nExec:
        pass

    return results
```

Listing A.2: Control del multiproc.

Es crea una llista compartida entre processos que serà utilitzada per les distintes execucions per emmagatzemar els resultats. També és crea un *mutex* per evitar concurrències a l'hora d'utilitzar-la.

S'inicialitzen[18] els diferents processos a executar, cada un d'ells amb els atributs requerits, annex A.6. El valor de la llavor del generador pseudoaleatori serà diferent per cada una d'aquestes execucions, incrementant-lo de manera seqüencial.

Com a màxim s'executaran tants processos com nombre de nuclis té el processador utilitzat, per controlar-ho s'utilitzarà un semàfor. Un procés s'executarà només si hi ha un testimoni disponible.

Els processos individuals queden definits a la funció **exeProcess(***algo, function, s, ite, mutex, semaphore, results***)**, codi A.3. En aquest és crea un objecte que representa l'execució d'un determinat algoritme i executa el procés de cerca cridant el seu mètode **run()**, un cop l'execució ha finalitzat afegeix aquest objecte amb els resultats a la llista compartida assegurat-se que no l'està utilitzant un altre procés. Desprès allibera un testimoni del semàfor i finalitza l'execució.

```
def exeProcess(algo, function, s, ite, mutex, semaphore, results):
    result = algo(function, seed = s,iterations = ite)
    result.run()

    mutex.acquire()
    results.append(result)
```

[17] `http://docs.python.org/library/multiprocessing.html`
[18] no confondre amb executar

```
    mutex.release()

    semaphore.release()
```

Listing A.3: Execució d'un procés

Un cop executats tots els processos es generen els objecte que contindran els estadístics per cada parell algoritme funció. Els atributs d'aquests objectes estan representats a la classe *statsResults* de la figura A.1.

Donat que el temps d'execució pot arribar a ser llarg, s'ha inclòs una barra de progrés per tal d'anar informant de l'estat de l'execució, figura A.3.

```
COMPARATIVA ABC vs GA, DE, PSO
------------------------------
25% ■■■■■■■■■■□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ 6 segons
Optimitzant funció: colville
Algortime utilitzat: PSO
```

Figura A.3: Barra de progrés de l'execució

Si s'ha seleccionat l'opció per desar un fitxer amb l'informació, el següent punt de l'execució consisteix a generar-lo, per a tal fi és genera un diccionari que contingui els resultats dels estadístics i es desa al disc utilitzant la llibreria *shelve* de Python[19].

El següent pas consisteix a, si s'ha especificat, mostrar els estadístics per a cada funció dels 4 algoritmes. Es generà una taula amb els resultats utilitzant la funció **resultsTable()** i es mostrà per pantalla utilitzant la funció **pprint_table(***table***)**. De la manera similar s'actuarà si es volen els resultats en format LaTeX, cridant la funció **latexOutput(***table, nom_del_fitxer***)** es generaran els fitxers corresponents.

Funció	Mínim		GA	DE	PSO	ABC
sphere	0	Fitness mean	4.59774353407e-05	0.0	0.0	0.0
		Fitness stdDev	3.55088544306e-05	0.0	0.0	0.0
		Fitness SEM	6.48300018761e-06	0.0	0.0	0.0
		Min Fitness	6.07997525253e-06	0	0	0
		Max Fitness	0.00013963683736	0	0	0
		Success executions percent	0.00%	100.00%	100.00%	100.00%
		Iterations mean	-	842.838709677	1376.0	1022.19354839
		Iterations stdDev	-	23.7881737685	27.1542784605	36.1987518541
		Iterations SEM	-	0.0197769318793	0.0448836100881	0.0358996303199
		Min Iterations	-	802	1322	956
		Max Iterations	-	937	1425	1095
		Duration mean	-	4.71609653196	13.0680799715	5.52230028183
		Duration stdDev	-	0.108322717086	0.245837657075	0.196630373323
		Duration SEM	-	0.0197769318793	0.0448836100881	0.0358996303199
		Min Duration	-	4.4853579998	12.5440921783	5.17595505714
		Max Duration	-	5.04827785492	13.6472461224	5.92478203773

Figura A.4: Estadístics de la funció Sphere

Un dels punts que resten per completar l'execució és realitzar la comparativa entre els ABC i els altres 3 algoritmes. Aquest és fa mitjançant la funció **compare**:

[19] `http://docs.python.org/library/shelve.html`

```
def compare(alg1,alg2,resultsDic,significance):
    functions = ['sphere','step','sumSquares','colville','zakharov','schwefel2p22',\\
        'himmelblau','rastrigin','schwefel','michalewicz5','schaffer','shubert',\\
        'griewank','ackley','shekel10']
    comparations = []


    for f in functions:
        test = stats.ttest_ind(resultsDic[alg1][f]['stats'].best,\\
                resultsDic[alg2][f]['stats'].best)
        comparations.append(comparationResult(alg1, alg2, f, test[0], test[1]))

    comparations.sort(key=attrgetter('p_value'),reverse=True)

    for i in range(len(comparations)):
        comparations[i].rank = i + 1
        comparations[i].significance = significance/comparations[i].rank
        comparations[i].hValidation()

    return comparations
```

Listing A.4: Funció que compara l'eficàcia de 2 algoritmes

Aquesta funció realitza la comparativa utilitzant mètode *t-test* per a mostres independents implementat dins la llibreria SciPy. Com s'ha comentat al capítol que descriu l'experiment es tindrà en compte la correcció de Bonferroni per a realitzar-la.

Per últim es mostren els resultats i generen els fitxes LaTeXutilitzant un procés similar a l'empleat per mostrar els estadístics. A la figura A.5 es pot veure un exemple.

COMPARATIVA ABC vs DE:

Funció	t-value	p-value	Ranquing	Significancia	Millor algoritme
sphere	1.0	0.321326503293	1	0.05	-
step	1.0	0.321326503293	2	0.025	-
sumSquares	1.0	0.321326503293	3	0.0166666666667	-
zakharov	1.0	0.321326503293	4	0.0125	-
schwefel2p22	1.0	0.321326503293	5	0.01	-
booth	1.0	0.321326503293	6	0.00833333333333	-
ackley	1.0	0.321326503293	7	0.00714285714286	-
michalewicz5	-1.2423193901	0.218952507822	8	0.00625	-
colville	-1.55155286313	0.126028427342	9	0.00555555555556	-
griewank	-1.78125962328	0.079931643496	10	0.005	-
shubert	3.99960270734	0.000176555903096	11	0.00454545454545	DE
schaffer	-4.38829833925	4.69901763404e-05	12	0.00416666666667	ABC
shekel10	13.2671439429	1.69287652026e-19	13	0.00384615384615	DE
schwefel	-19.3406369125	1.82087429887e-27	14	0.00357142857143	ABC
rastrigin	-23.3141713357	9.03959829646e-32	15	0.00333333333333	ABC

Figura A.5: Resultat d'una comparativa

A.5 Implementació de les funcions d'avaluació

Les funcions d'avaluació queden representades per la classe *Function*, implementada a l'arxiu *functions.py*. Els seus atributs són els paràmetres que configuren la cerca:

dimensions, interval, mínim teòric... També conté un atribut, **evalf**, que representa l'implementació de la formula matemàtica que la defineix. Aquesta funció pot ser un atribut en lloc d'un mètode ja que per a Python, tots els elements són objectes i per tant poden ser assignats a una variable.

Les implementacions de les formules matemàtiques també es troben en aquest arxiu i tenen la particularitat d'utilitzar un patró *decorador* [25, pag. 175] per implementar una memòria cache que permeti reduir el temps necessari, emmagatzemant per a tal fi, els càlculs més recents i en cas de tornar ser requerits no haver-los de repetir.

```
@decorator
def memoize(f):
    """Decorador que implementa una cache dels darrers 20000 calculs realitzats"""
    maxsize = 20000
    current = [0]
    cache = {}  # map from key to value
    heap = []   # list of keys, in LRU heap
    """decorator to 'memoize' a function - caching its results"""
    def _f(*args):
        key = tuple(args[0])
        if not key in cache:
            if current[0] < maxsize:
                current[0] += 1
            else:
                # pop oldest element
                (_,oldkey) = heapq.heappop(heap)
                cache.pop(oldkey)
            # insert this element
            cache[key] = f(*args)
            heapq.heappush(heap,(time.time(),key))
        return cache[key]
    return _f
```

Listing A.5: Decorador memoize

A.6 Implementació dels EC

Els capítols descriptius d'aquesta memòria s'han detallat alguns punts comuns entre els algoritmes computació evolutiva. Aprofitant aquest factor per reduir duplicitats, s'ha implementat un classe *Ec* de la qual heretaran atributs i mètodes els algoritmes utilitzats a l'experiment (figura A.1), de manera similar a l'exposada al patró *estratègies* definit per Gamma [25, pag. 315].

Els atributs d'aquesta classe fan referència a:

- **function**: funció d'avaluació utilitzada.
- **seed**: la llavor de generador pseudoaleatori utilitzada.
- **iterations**: nombre d'iteracions màximes permeses.
- **threshold**: error permès en el càlcul del mínim.

- **nPop**: Nombre de solucions candidates.
- **bestIndv**: Paràmetres de la millor solució trobada fins el moment.
- **bestFitness**: Avaluació de la millor solució trobada fins el moment.
- **startTime**: Temps d'inici de l'execució.
- **stopTime**: Temps de fi de l'execució.
- **iterationsUntilGoal**: nombre d'iteracions utilitzades.
- **isGoalFounded**: variable que indica si l'execució a finalitzat amb èxit o no.

També s'implementen quatre mètodes que són d'utilitat:

- **initPopulation()**: La representació de les solucions candidates la mateixa per a tots els algoritmes exceptuant els PSO. S'inicia una llista amb tants de valors com dimensions hi hagi, i a cada valor se li assigna un nombre dins l'interval determinat de manera uniformement aleatòria, codi A.6.
- **getTotalTime()**: Retorna el temps utilitzat per a l'execució restant el temps de fi al temps d'inici.
- **setBestIndv(*List*)**: determina el millor individu de una llista de solucions candidates.
- **updateResults(*List*)**: donada una llista amb solucions candidates, determina la millor i, si procedeix, actualitza les dades corresponents: millor solució, fi amb èxit, nombre d'iteracions... Codi A.7.

```
def initPopulation(self):
        random.seed(self.seed)#Iniciam el generador pseudoaleatori
        return [[random.uniform(self.function.rangeA, self.function.rangeB)\\
        for d in range(self.function.dimensions)] for n in range(self.nPop)]
```

Listing A.6: Mètode que inicialitza una població de solucions candidates

```
def updateResults(self, population):
        populationBest = min(population, key=self.function.evalf)
        if self.function.evalf(populationBest) < self.bestFitness:
                self.bestIndv = populationBest
                self.bestFitness = self.function.evalf(populationBest)

        self.iterationsUntilGoal += 1

        if self.bestFitness < self.function.minimum+self.threshold:
                self.bestFitness = self.function.minimum
                self.isGoalFounded = True
```

Listing A.7: Mètode que actualitza els resultats

A.6.1 Implementació del GA

L'implementació dels GA es fa a través de la classe homònima *Ga*. Aquesta classe té uns quants atributs específics:

- **mutationRate**: probabilitat que una dimensió sofreixi una mutació.
- **eliteRate**: percentatge de membres de la generació progenitora que poden passar a la següent generació.
- **population**: població de les solucions candidates.

Els diferents mètodes implementats representen les funcionalitats requerides per aquests tipus d'algoritmes ja definides a l'apartat 3.1.

El mètode **run()** és el que controla l'execució de l'algoritme i conté el bucle principal on es duu a terme el procés iteratiu.

El primer objectiu és seleccionar els pares encarregats de produir la nova generació. El mètode **parentsSelection()** implementa aquest comportament mitjançant la selecció per rànquing i el mostreig SUS.

```
def parentSelection(self):
        popRanked = sorted(self.population, key=self.function.evalf, reverse=True)

        probSelection = (lambda nPop, rank, s = 1.5:\\
         ((2-s)/nPop)+((2*rank*(s-1))/(nPop*(nPop-1))))

        ranking = [[probSelection(len(self.population),rank),popRanked[rank]]\\
        for rank in range(len(self.population))]

        matingPool = []
        r = random.uniform(0,(1/len(self.population)))
        sumProb = 0
        i = 49

        while len(matingPool) < len(self.population):
                sumProb += ranking[i][0]
                while r <= sumProb:
                        matingPool.append(ranking[i][1])
                        r += 1.0/len(self.population)
                i -= 1
        return matingPool
```

Listing A.8: Mètode de selecció paterna

Un cops seleccionats els pares, es procedeix a realitzar el *crossover* per tal de generar els nous individus. Aquesta tasca es portada a terme pel mètode **corssover(***matingPool***)** el qual implementa el descrit a l'apartat 3.1.4.

Generat l'*offspring*, s'ha d'aplicar el procés de mutació als nous individus. El mètode **mutation(***offspring***)** implementa aquesta funcionalitat:

```
def mutate(self,offspring):
        for n in range(len(offspring)):
                for gen in range(len(offspring[n])):
                        if random.random() < self.mutationRate:
                                offspring[n][gen] = \\
                                random.uniform(self.function.rangeA,\\
                                self.function.rangeB)
```

Listing A.9: Mètode utilitzat per mutar els individus

Arribat aquest punt, només resta determinar quins seran els individus que formaran la nova generació tenguen en compte l'estratègia elitista, apartat 3.1.6, i actualitzar els resultats utilitzant el mètode **updateResults(*list*)** anteriorment descrit.

A.6.2 Implementació del DE

La classe De[20] és la que implementa el funcionament de l'algoritme DE. Els atributs d'aquesta classe, a part dels heretats de la classe Ec, són:

- **f**: representa el factor d'escalat utilitzat. Per defecte 0.4717.
- **cr**: probabilitat de seleccionar la dimensió del vector mutant a l'hora de realitzar el *crossover*. Per defecte 0.8803
- **population**: conjunt de solucions candidates

Aquesta classe només inclou un mètode, **run()**, el qual, es l'encarregat d'executar l'algoritme. El pes del procés de cerca recau principalment sobre l'operador de mutació:

```
for indv in self.population:
        vectorsMut = random.sample(self.population,3)
        while indv in vectorsMut:
                vectorsMut = random.sample(self.population,3)
        mutation = [(vectorsMut[0][i]+self.f*(vectorsMut[2][i]-vectorsMut[1][i]))\\
         for i in range(self.function.dimensions)]
```

Listing A.10: Mutació en els DE

Per a cada individu de la població es seleccionen 3 membres distints del conjunt de solucions candidates i és combinen per formar una solució mutant, *mutation*. Un cop s'ha generat aquest vector mutant, es genera el nou individu combinant-lo amb l'individu original:

```
for d in range(self.function.dimensions):
        r = random.random()
        if r <= self.cr:
                if mutation[d] < self.function.rangeA:
                        offspring.append(self.function.rangeA)
                elif mutation[d] > self.function.rangeB:
                        offspring.append(self.function.rangeB)
```

[20] Arxiu adjunt *algorithms/de.py*

```
            else:
                    offspring.append(mutation[d])
        else:
            offspring.append(indv[d])
```

Listing A.11: Crossover en els DE

Sa de remarcar, que els valors de les dimensions del vector mutant poden estar fora dels límits definits per l'interval de cerca, per aquest motiu, es realitza una comprovació a l'hora de generar el nou membre per tal de respectar aquests límits. Un cop fet això per a tots els membres de la població, es pot procedir a actualitzar els resultats i comprovar si es satisfan les condicions de fi.

A.6.3 Implementació del PSO

L'implementació de l'algoritme PSO és un tant més complexe que les vistes fins ara. Això es deu principalment al fet que utilitza una representació diferent per els individus de la població: les partícules. Per tant en aquest cas s'utilitzaran dues classes.

Implementació de les partícules

Una partícula queda definida per la classe Particle la qual té els següents atributs:

- **currentPos**: Posició de la partícula dins l'espai de possibles solucions.
- **lbest**: Millor posició aconseguida per la partícula.
- **nbest**: Millor posició aconseguida per algun membre del conjunt de veïns
- **neighborhood**: Conjunt de veïns.
- **velocity**: Velocitat de la partícula.

A més una partícula té 2 mètodes:

- **findNBest()**: Comprova la millor posició aconseguida per els veïns i actualitza l'atribut *nbest* conseqüentment. Aquest mètode es utilitzat a l'hora d'inicialitzar la població.
- **setLbest(***List, Float***)**: Mètode que actualitza l'informació de la millor posició aconseguida fins el moment i transmet aquesta informació als veïns per tal que, si procedeix, actualitzin l'informació de millor posició del conjunt de veïns.

```
def setLbest(self, lb, lbScore):
    self.lbest['pos'] = lb[:]
    self.lbest['score'] = lbScore
    if lbScore < self.nbest['score']:
        self.nbest['pos'] = lb[:]
        self.nbest['score'] = lbScore
            if self.neighborhood[0].nbest['score'] > lbScore:
                self.neighborhood[0].nbest['pos'] = lb[:]
```

```
            self.neighborhood[0].nbest['score'] = lbScore
        if self.neighborhood[2].nbest['score'] > lbScore:
            self.neighborhood[2].nbest['pos'] = lb[:]
            self.neighborhood[2].nbest['score'] = lbScore
```

Listing A.12: Mètode selfLbest

Implementació de la classe PSO

La classe Pso[21] representa l'execució de un algoritme PSO. Els atributs que la composen són:

- **population**: Població de partícules
- **chi**: Constant utilitzada per calcular la constricció simple de Clerk. Valor per defecte, 0.73
- **phiP**: Influència de la millor posició aconseguida per la particula. Valor per defecte 2.05
- **phiG**: Influència de la millor posició aconseguida pel conjunt de veïns. Valor per defecte 2.05

En quan els mètodes, n'hi ha dos, el primer **initPopulation()** redefineix el mètode homònim heretat de la classe *Ec* per tal d'adaptar-lo a les peculiaritats de la representació de la població:

```
def initPopulation(self):

    random.seed(self.seed)

    #Velocitat max. inicial de les part.
    vMax = (self.function.rangeB-self.function.rangeA)*0.15
    pop = [Particle(self.function.rangeA, self.function.rangeB, \\
            self.function.dimensions, vMax) for x in range(self.nPop)]

    # Iniciam neighborhood
    for i in range(len(pop)):
            pop[i].neighborhood = [pop[(i-1)%self.nPop],pop[i],\\
                    pop[(i+1)%self.nPop]]

    # Determinam les millor posicions coneguedes
    for p in pop:
            p.lbest['pos'] = p.currentPos[:]
            p.nbest['pos'] = p.currentPos[:]
            evalCurrent = self.function.evalf(p.currentPos)
            p.lbest['score'] = evalCurrent
            p.nbest['score'] = evalCurrent

    for p in pop:
            p.findNBest()
```

[21] Arxiu adjunt *algorithms/pso.py*

```
return pop
```

Listing A.13: Mètode initPopulation especific per l'algoritme PSO

Com en la resta d'implementacions fins ara mostrades, el control de l'execució recau a al mètode *run()*. El procés de cerca de la solució ve determinat per els canvis en la posició de les partícules tenguen en compte influències pròpies i externes. L'implementació del càlcul de la nova posició, per a una determinada partícula, es el que és pot veure tot seguit:

```
for d in range(self.function.dimensions):
        randP = random.uniform(0,self.phiP)
        randG = random.uniform(0,self.phiG)
        p.velocity[d] = self.chi*((p.velocity[d]) + \\
                randP*(p.lbest['pos'][d]-p.currentPos[d]) + \\
                randG*(p.nbest['pos'][d]-p.currentPos[d]))

        p.currentPos[d] = p.currentPos[d] + p.velocity[d]

        #Limitam la pos. al rang de cerca
        if p.currentPos[d] > self.function.rangeB:
                p.currentPos[d] = self.function.rangeB
        elif p.currentPos[d] < self.function.rangeA:
                p.currentPos[d] = self.function.rangeA
```

Listing A.14: Càlcul de la nova posició

Com en el cas de l'implementació de l'algoritme DE, es controla que la posició de la partícula, un cop actualitzada, estigui dins el rang establert per a la cerca de la solució, si no és així, ho corregeix. Un cop actualitzada la posició es comprova que si aquesta representa la millor posició aconseguida per la partícula fins aquest moment. Si és així, actualitza els atributs de la partícula i informa als seus veïns d'aquest esdeveniment:

```
#Avaluam la nova pos.
posFitness = self.function.evalf(p.currentPos)

#Actualitzam si procedeix les dades de millor pos. aconseguida.
if posFitness < p.lbest['score']:
        p.setLbest(p.currentPos[:],posFitness)
        if posFitness < bestValue:
                bestValue = posFitness
                gbest = p.currentPos[:]
```

Listing A.15: Avaluació de la nova posició

Igual que la resta d'algoritmes, un cop actualitzada la posició per a cada individu, s'actualitzen els resultats de l'execució i es comprova si es compleixen les condicions de fi.

A.6.4 Implementació del ABC

Seguint el mateix procediment que s'acaba d'aplicar a les darreres seccions, es començarà mostrant els atributs que descriuen la classe Abc la qual implementa l'algoritme de mateix nom. En aquest cas, només hi ha un atribut, **foodSrc**. Una llista de les solucions candidates equivalent al que fins ara s'havia anomenat **population**.

Com en casos anteriors, el mètode **run()** és l'encarregat d'iniciar el procés iteratiu de cerca descrit a l'apartat 3.4.1. Cada iteració, pot ser dividida en quatre fases:

1. Fase d'abelles *Ocupades*
2. Fase d'abelles *desocupades*
3. Actualització dels resultats
4. Fase d'abelles *escolta*

La primera fase, que podem veure al Listing A.16, genera per a cada solució candidata un versió desplaçada de la mateixa utilitzant la funció **generateV(***List***)** que es descriurà tot seguit. Si l'adaptabilitat de la nova versió és millor que l'anterior[22], aquesta passa a ocupar el lloc de l'antiga i es posa el comptador del limit a zero. Si no és així, es descarta la nova versió i s'incrementa el comptador.

```
for i in range(len(self.foodSrc)):
        v = self.generateV(i)

        if self.fit(v) > self.fit(self.foodSrc[i]):
                self.foodSrc[i] = v[:]
                foodTrialCount[i] = 0
        else:
                foodTrialCount[i] += 1
```

Listing A.16: Fase d'abelles ocupades

El mètode **generateV(***List***)** descrit al Listing A.17, és l'encarregat de generar el nou vector desplaçat. Aquest es genera trasformant una dimensió, seleccionada de manera no determinista, amb relació a la mateixa dimensió d'una altre posició del conjunt de solucions. Com en casos anteriors, es controlarà que la nova posició estigui dins els límits establerts per a la cerca.

```
def generateV(self, i):
        j = random.choice(range(self.function.dimensions))
        k = i
        while k == i:
                k = random.choice(range(self.nPop))
        #Factor d'escalat
        phi = random.uniform(-1,1)
        v = self.foodSrc[i][:]
        #Calcul del despl.
```

[22]Calculada mitjançant la funció **fit(***List***)** per els motius descrits a l'apartat 3.4.1.

```
        newParameter = v[j] + phi*(v[j]-self.foodSrc[k][j])
        #Limitam els resultats dins els rangs desitjats
        if newParameter < self.function.rangeA:
                newParameter = self.function.rangeA
        elif newParameter > self.function.rangeB:
                newParameter = self.function.rangeB
        v[j] = newParameter
        return v
```

Listing A.17: Generador de desplaçament

La fase d'abelles desocupades està implementada tal com mostra el Listing A.18. Primer es genera una llista que conté posicions d'aliment seleccionades segons la seva aptitud[23] utilitzant el mètode **onlookerFoodSelection()**. Després s'itera sobre tots els elements d'aquesta llista de manera anàloga a l'aplicada a la fase d'abelles ocupades.

```
#Dest. de les abelles desocupades
onlookerFoodDestination = self.onlookerFoodSelection()

for i in onlookerFoodDestination:
        v = self.generateV(i)

        if self.fit(v) > self.fit(self.foodSrc[i]):
                self.foodSrc[i] = v[:]
                foodTrialCount[i] = 0
        else:
                foodTrialCount[i] += 1
```

Listing A.18: Fase abelles desocupades

El mètode **onlookerFoodSelection()** és l'encarregat de retornar un llistat de posicions ponderant el nombre d'aparicions d'aquestes en base a la seva *fitness*. S'utilitza el mateix algoritme que en els GA, SUS, per omplir la llista.

```
def onlookerFoodSelection(self):
        #Probabilitat de ser seleccionat per una abella no ocupada
        totalFit = sum([self.fit(food) for food in self.foodSrc])
        p = [self.fit(food)/totalFit for food in self.foodSrc]
        #Omplint els resultats utilitzant el SUS
        onlookerFoodDestination = []
        r = random.uniform(0,(1/len(self.foodSrc)))
        sumProb = 0
        i = len(self.foodSrc)-1

        while len(onlookerFoodDestination) < len(self.foodSrc):
                sumProb += p[i]
                while r <= sumProb:
                        onlookerFoodDestination.append(i)
                        r += 1.0/len(self.foodSrc)
                i -= 1
        return onlookerFoodDestination
```

Listing A.19: Selecció de les posicions de les abelles desocupades

[23] Més presència de les que tenen millor aptitud.

Arribat a aquest punt, s'actualitzen els resultats de l'execució i és comprova si s'ha arribat a alguna condició de fi. Si no és així, s'executa el que s'anomena fase d'abelles escolta. Aquesta fase consisteix a comprovar si alguna posició a superat el nombre d'iteracions sense millorar la seva aptitud, si és així, la elimina i en genera una altre de manera aleatòria.

```
for i in range(len(foodTrialCount)):
        if foodTrialCount[i] > limit:
                self.foodSrc[i] = [random.uniform(self.function.rangeA,\\
                        self.function.rangeB)\\
                        for d in range(self.function.dimensions)]
                foodTrialCount[i] = 0
```

Listing A.20: Fase d'abelles escolta

Glossari

codi genètic Conjunt de normes que permeten la conversió de seqüències d'àcids nucleics (ADN o ARN) en proteïnes dins les cèl·lules dels éssers vius. 7, 13, 64

computació evolutiva Branca de la intel·ligència artificial que involucra problemes d'optimització combinatoria i s'inspira en els mecanismes de l'evolució biológica. 5, 6, 10, 17, 20, 21, 50, 56, 64

cromosoma Orgànul filiforme format per cromatina present en el nucli cel·lular dels organismes eucariotea que porta la dotació genètica. 64

estocàstic Relatiu o pertanyent a una successió d'esdeveniments que depenen de l'atzar. 10, 16, 20, 21, 64

Eusocialitat és una forma de vida col·lectiva social que es caracteritza per generacions que s'encavalquen; cura de les cries de manera cooperativa i castes especialitzades reproductives i no reproductives. 37, 64

fenotip Conjunt de caràcters visibles que un organisme presenta com a resultat de la interacció entre el seu genotip i l'ambient. 11, 13, 64

gen Fragment de cromosoma en què es troba codificada una unitat d'informació genètica. 11, 64

genotip Informació genètica total d'un organisme o constitució genètica d'un organisme. 11–13, 31, 64

genètica Branca de la biologia que estudia els fenòmens de l'herència i de la variació de les espècies. 5, 11–13, 64

mineria de dades La mineria de dades (Data Mining) és un procés no trivial d'identificació d'informació vàlida potencialment útil i entenible de patrons comprensibles que es troben ocults en les dades. 17, 64

mutació Les mutacions són canvis permenents en l'ADN que es transmeten a la descendència que poden ésser causats per agents exògens o endògens. 31–33, 57, 64

Llistat d'acrònims

ABC Artificial Bee Colony. 5, 6, 8, 11, 20, 25, 38, 54, 64

ADN Àcid desoxiribonucleic. 12, 64

DE Differential Evolution. 5, 7, 10, 20, 25, 34, 59, 62, 64

EA Evolutionary Algorithms. 10, 11, 13, 15, 64

FPS Fitness Proportional Selection. 28, 29, 64

GA Genetic Algorithm. 5, 7, 10, 16, 18–20, 25, 27, 28, 32, 40, 57, 64

MBF Mean Best Fitness. 20, 64

PFC Projecte Final de Carrera. 5, 6, 17, 20, 28, 32, 64

PSO Particle Swarm Optimization. 5, 8, 11, 20, 25, 34, 57, 60, 61, 64

SGA Simple Genetic Algorithm. 28, 30–32, 64

SI Swarm Intelligence. 11, 16, 64

SUS Stochastic Universal Sampling. 29, 40, 58, 64

Bibliografia

[1] Wikipedia - boole's inequality, Abril 2012.

[2] Wikipedia - evolutionary computing, Abril 2012.

[3] Wikipedia - evolutionary programming, Abril 2012.

[4] Wikipedia - himmelblau's function, Abril 2012.

[5] D.J. Ackley. *A connectionist machine for genetic hillclimbing.* Klower Academic Publishers, 1987.

[6] Miguel Ratòn Almanso. Optimización continua basada en algoritmos de estimación de regresión. *Universidad Politécnica de Madrid - PFC.*

[7] J.D. Bagley. The behavior of adaptive systems which employ genetic and correlation algorithms. 1967.

[8] Josep Gibergans Bàguena, Àngel J. Gil Estallo, and Carles Rovira Escofet. *Estadística.* FUOC, quarta edició edition, febrer 2009.

[9] J.E. Baker. Reducing bias and inefficiency in the selection algorithms. 1985.

[10] G. Bateson. *Mind and nature: A necessary unity.* Bantam Books New York, 1980.

[11] G. Beni and J. Wang. Swarm intelligence in cellular robotic systems. *Robots and Biological Systems: Towards a New Bionics?*, pages 703–712, 1993.

[12] H.J. Bremermann. Optimization through evolution and recombination. *Self-organizing systems*, pages 93–106, 1962.

[13] Martin Brookes. *Get a Grip on Genetics.* Time Life Education, 1 edition edition, 1998.

[14] M. Clerc and J. Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58–73, 2002.

[15] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142, 1991.

[16] C. Darwin. *The origin of species.* Collier, 1937.

[17] L.N. de Castro and J. Timmis. An artificial immune network for multimodal function optimization. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 1, pages 699–704. IEEE, 2002.

[18] K. Deep and M. Thakur. A new mutation operator for real coded genetic algorithms. *Applied Mathematics and Computation*, 193(1):211–230, 2007.

[19] R.C. Eberhart, Y. Shi, and J. Kennedy. *Swarm intelligence.* Elsevier, 2001.

[20] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing.* Springer Verlag, 2003.

[21] L.J. Fogel. Evolutionary programming in perspective: The top-down view. *Computational Intelligence: imitating life*, pages 135–146, 1994.

[22] L.J. Fogel, A.J. Owens, and M.J. Walsh. Artificial intelligence through a simulation of evolution. In *Biophysics and Cybernetic systems. Proc. of the 2nd Cybernetic Sciences Symposium*, pages 131–155, 1965.

[23] A.S. Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences*, 10:484–491, 1957.

[24] A.A. Freitas. *Data mining and knowledge discovery with evolutionary algorithms.* Springer Verlag, 2002.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995.

[26] D.E. Goldberg. Computer-aided gas pipeline operation using genetic algorithms and rule learning. 1983.

[27] D.E. Goldberg and C.H. Kuo. Genetic algorithms in pipeline optimization. In *PSIG Annual Meeting*, 1985.

[28] R.L. Haupt, S.E. Haupt, and J. Wiley. *Practical genetic algorithms.* Wiley Online Library, 2004.

[29] J.H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

[30] J.H. Holland. *Adaptation in natural and artificial systems.* Number 53. University of Michigan press, 1975.

[31] D. Karaboga. An idea based on honey bee swarm for numerical optimization. *Techn. Rep. TR06, Erciyes Univ. Press, Erciyes*, 2005.

[32] D. Karaboga and B. Akay. A comparative study of artificial bee colony algorithm. *Applied Mathematics and Computation*, 214(1):108–132, 2009.

[33] AJ Keane. The design of a satellite beam with enhanced vibration performance using genetic algorithm techniques. *The Journal of the Acoustical Society of America*, 99(4):2599–2603, 1996.

[34] J. Kennedy. Thinking is social. *Journal of Conflict Resolution*, 42(1):56–76, 1998.

[35] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[36] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1671–1676. Ieee, 2002.

[37] J.R. Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[38] Z. Michalewicz. *Genetic algorithms+ data structures*. Springer, 1996.

[39] Marcin Molga and Czeslaw Smutnicki. Test functions for optimization needs. 2005.

[40] M.E.H. Pedersen. Good parameters for differential evolution. Technical report, Technical report, Hvass Computer Science Laboratories, 2010.

[41] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dr.-Ing.* PhD thesis, Thesis, Technical University of Berlin, Department of Process Engineering, 1971.

[42] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2010.

[43] H.P. Schwefel. *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

[44] R. Storn and K. Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[45] V. Tereshko. Reaction-diffusion model of a honeybee colonys foraging behaviour. In *Parallel Problem Solving from Nature PPSN VI*, pages 807–816. Springer, 2000.

[46] A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[47] A.M. Turing. Intelligent machinery (1948). *B. Jack Copeland*, page 395, 2004.

[48] X.S. Yang and S. Deb. Cuckoo search via lévy flights. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214. IEEE, 2009.